**The Report Committee for Jinxiang Lu**
**Certifies that this is the approved version of the following report:**

# QUARTS: A Quantitative Research and Trading System

## COMMITTEE:

Adnan Aziz, Supervisor

Thomas J. Graser

# QUARTS: A Quantitative Research and Trading System

**by**

**Jinxiang Lu, B.E.;M.E.**

## Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

## Master of Science in Engineering

**The University of Texas at Austin**

**May, 2013**

For

Cindy

## Acknowledgements

I would like to thank Professor Adnan Aziz, my supervisor, for giving me guidance and passion on this project and on the report. I would also like to thank PhD Thomas Graser, my reader, for providing a good approach to design software architecture systematically. Finally, I would like to thank my parents and my family for their support.

# Abstract

## QUARTS: A Quantitative Research and Trading System

Jinxiang Lu, MSE

The University of Texas at Austin, 2013

Supervisor: Adnan Aziz

This report presents a quantitative research and trading system (QUARTS) for US equities. After introduction of US stock market structure, it presents the quantitative model concept, specifically, its components and its interactions with different environments. Equipped with a software architecture design discipline that follows three steps — define the problem; design the solution; and deploy to sites — it designs the architecture of QUARTS. This is followed by a prototype implementation of research environment. Finally it gives two sample quantitative models to demonstrate the use of research environment. The report includes a detailed survey of Software Architecture and Design Methodologies to help readers to better understand the derivation of QUARTS architecture.

# Table of Contents

# List of Tables

# List of Figures

# Introduction

US stock market remained unchanged for centuries. It was revolutionized with the help of information technology in the 1980s. The new, fully connected, electronic stock market gave rise of automated quantitative computer models to trade stocks.

Although Java was used for some quantitative trading models, most of the high-frequency production-bound systems were written in C++ [1]. A very efficient language, C++ is suitable for implementing a fast system, but it is not the best language for quantitative researchers (quants) to design trading models. C++ requires a lot of details on implementations. A quant's time is better spent on modeling than programming especially because quants usually work on hundreds of models out of which only limited few are good enough to put into production. Even paired with dedicated quant developers, the current mode still slows time to market significantly. To make things worse, many quantitative models are only effective for a period of time in the market before unavoidable retooling.

The mixture of C++ as framework and an efficient script language as model should be able to solve the conflict between performance and productivity. Python is such a script language. It provides CAPI interfaces to interact with C efficiently. Moreover, one of interpreter implementation, Pypy, supports Just-In-Time (JIT) compiler which increases Python speed more than three times faster than standard interpreters.

QUARTS, a quantitative research and trading system, is designed as a mixture of C++ and Python. Mostly implemented in C++, QUARTS embeds an interpreter to run quantitative models written in Python language. All the models are designed, debugged and partially tested in Python environment with support from scientific, finance and data mining community. 2D and 3D plotting support in Python also boosts productivity of quants' research.

QUARTS also designs a workflow to help quants to run the same model in different environments. A model designed and partially tested in Python environment goes through a back-test environment written in C++ to be thoroughly tested with years of historical market data and optimized with an array of different options and parameters. Equipped with more accurate statistical results in a bigger sample space, quants put the model with optimal set of options and parameters into a forward-test environment with real-time market data and execution simulator to measure the effectiveness against current stock market. Finally the fine-tuned model runs in live trading environment with allocated capital. The model is thus switched to operational mode with traders actively monitoring the performance and scaling the model up or down as necessary.

Compared with pure C++ trading system, the loss of performance of QUARTS resides in evaluation of models written in Python. A trading system spends majority of time to process market data with ever-increasing volume. The model evaluation usually plays a small part in performance metrics. Even better, quantitative models are usually deployed for limited list of symbols. Thus evaluation takes even smaller portion of time of the whole system. It is estimated that a typical server with eight cores should be able to host 20-100 such models.

For some special models that require the fastest processing speed, the Python model can be rewritten in C++ easily as the environment that supports Python models can also support C++ models without any modification. In this case, QUARTS still saves quants a lot of time of design and back-test because only those performing models going into forward-test and production are required to be rewritten.

The report is organized as follows: We begin with the introduction of US stock market structure in Chapter 1. After that we analyze quantitative model in QUARTS and how it interacts with the environment in Chapter 2. Then we design the architecture of QUARTS in Chapter 3. Finally we describe the prototype implementation of the Python based research environment with two model examples in Chapter 4. The software architecture and design methodologies used in the report are provided in Appendix.

# Chapter 1: Stock Market Structure

This chapter provides the necessary details of US stock market structure to be used for analysis and design in later chapters. If you are already familiar with it, please proceed to the next chapter.

## TRADITIONAL STOCK MARKET STRUCTURE

Stock market has been around for centuries. It is originally a place when buyers and sellers meet to trade shares of public companies that are listed on exchanges. The exchange *trading floor* is the most common venue for traders to gather and traders working on the exchange floor are called *floor traders*. Usually trading floors are segmented into different trading units called *pits*, each of which serves a group of specific stocks. *Pit* is so called as the unit is circular in form and gets higher in altitude from center to the edge. The arrangement helps traders to see each other easily as originally the trade follows *open outcry* auction process where traders cry out their bids and asks with signs. In the middle of the *pit* are the book-keepers to record each trade. Trades are recorded and published in the form of a *tape*. The cash transfer among different trading accounts is done through financial clearing companies.

Traders can put up a *limit order* to buy or sell certain size of a specific stock at specific *limit price*. Or the trader can put up a *market order* to buy or sell certain size of a stock for the best price on the market at the moment. Traders can also *sell short* (or simply, *short*) the stock he or she does not have by borrowing from others and sell it to the market. Short sales are usually limit orders with price higher than the last trade price or at the last trade price if last trade price is higher than previous trade due to *uptick rule*. Short sellers can later *cover* the short position by buying the stock back.

Those who want to trade but cannot stay on the trading floor do so by calling up their *stock brokers*. A *broker* is a person or firm that facilitates trades between customers who does not assume any risk for the trade and charges a *commission*. Brokers have their

representative floor trader to act on behalf of their customers. Individuals can also trade directly with *dealers*. A *dealer* is a person or firm that buys and sells for his or her own inventory of securities and for others. A *dealer* therefore assumes risk for the transactions. *Dealers* may mark securities up or down to make a profit on their transactions.

The stock markets trade public companies. A public company is the one that sells some part of its shares to public through *initial public offering* (IPO) process. Public companies are highly regulated by *Securities Exchange Committee* (SEC) for the interest of general public. Public companies can also be traded through inter-dealer brokers system outside the exchange through a separate quote system. This is the original form of *NASDAQ*, *National Association of Securities Dealers Automated Quotations*. Companies not qualified to list on national exchanges can list and trade on OTC Bulletin Board (OTCBB). These issues are called *over-the-counter (OTC)* issues. Some OTC issues also register or report to SEC or other regulatory agencies while not regulated. Others not doing so are called *pink sheets* because of the color of the issue. Those public companies that do not meet the strict standards to list on national exchanges anymore are usually moved to *OTC* market unless they are bankrupted. (NASDAQ later obtained exchange status from SEC and can also list public companies.)

To stabilize the stock market, NYSE introduces stock *specialists* who are obliged to post bids and offers for the stock listed on NYSE they specialize in even if no one else bids or offers. In reality, specialists buy from sellers and sell to buyers frequently and maintain an inventory that is usually cleared at the end of the trading day. One stock can have only one specialist. NASDAQ, on the other hand, gives certain companies *market maker* status for stocks listed on its exchange. *Market makers* serve the same purpose as specialists but each stock can have multiple *market makers*. This stabilizes the stock quotes further. Nowadays *market making* is mostly through quantitative trading models.

Other than *SEC* as a federal agency to regulate and oversee the stock market, stock exchange and broker dealers also form *self regulatory organizations (SRO)* to

oversee its members. Originally NYSE has its own enforcement arm and NASDAQ has *National Association of Stock Dealers* (*NASD*). Later on both are merged into a new independent regulator, *Financial Regulatory Authority* (*FINRA*), the only one to regulate and oversee all US securities.

The traditional stock market structure before electronic trading (*e-trade*) becomes the norm is pretty straightforward. Figure 1 from [1] illustrates three tiers of services:

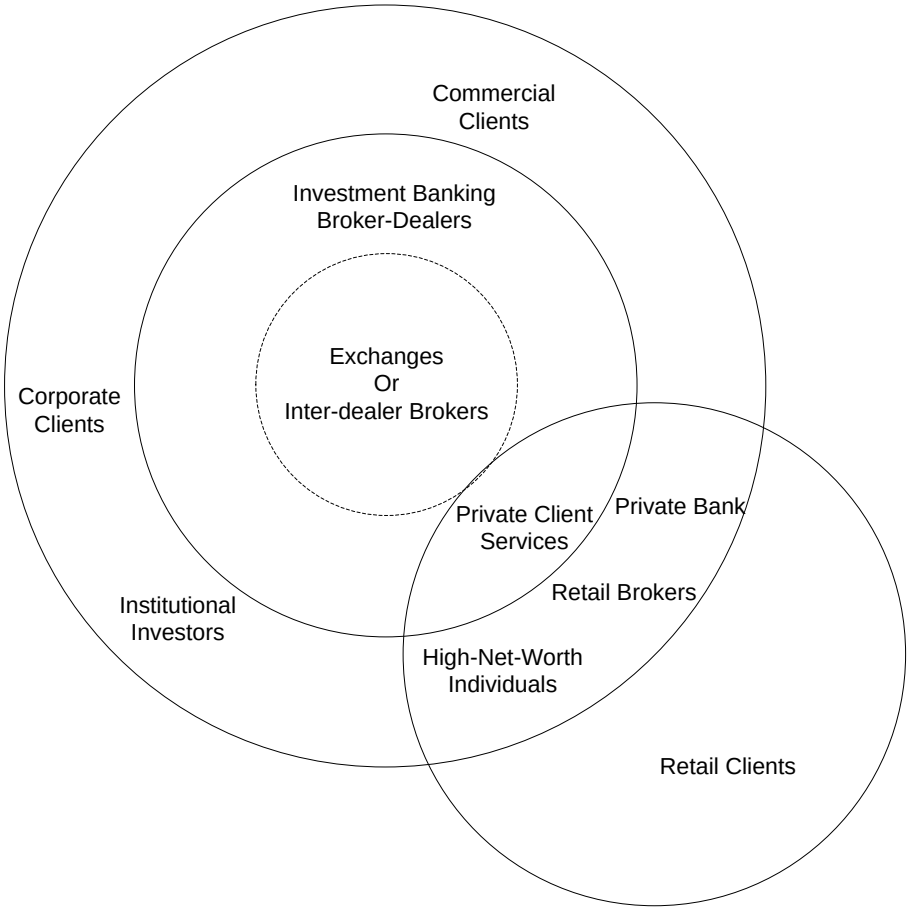


Figure 1: Traditional Stock Market Structure

- *Exchanges or Inter-dealer brokers*. Exchange is the place to list and trade public companies while Inter-dealer brokers' network is the one to trade over-the-counter (OTC) and OTC Pink Sheets companies as well as public companies.

- *Investment Banking Broker-Dealers.* The investment banks act as brokers that facilitate big clients' trade and earn commission. On the other hand, they buy stocks on their own and sell to public with a profit. Investment banks usually have their representatives on the trading floor of exchange to finalize the trade. They are also called *Prime Brokers* or *Prime Dealers.*
- *Retail Brokers*. Retail Brokers serve small clients for commissions. Retail Brokers usually redirect their clients' orders to Investment Banking Broker-Dealers.

Exchange and Inter-dealer Brokers provide most services to investment banks while also serve other individuals through their Private Client Services directly. Corporate Clients, Institutional Clients, Commercial Clients and Private Bank Clients and High-Net-Worth Individuals are typical big clients for Investment banks. Retail Clients goes through Retail Brokers. The right bottom circle contains different types of individuals accessing stock markets through tier one, tier two and tier three services.

**MODERN DAY STOCK MARKET STRUCTURE**

The advent of information technology revolutionized the stock market structure. One major revolution is the separation of exchange and market. Exchange is where the stock lists while market is where the stock trades. Accompanying a handful of exchanges in US are numerous electronic stock trading markets or venues. Each one provides special services to compete for trading volumes.

The markets outside exchange are called *Alternative Trading Systems (ATS).* Those ATSs acting exactly as markets in exchange are called *Electronic Communication Networks (ECN).* The big ECNs include Island, later merged into *INET* and now part of NASDAQ, Archipelago (*ARCA*), bought by *New York Exchange* (*NYSE*), *BATS (Best ATS),* now also an Exchange and Direct *EDGE*.

*ATSs* that do not post their quotes are called *Dark Pools*. *Dark Pools* are especially attractive to institutional clients as they usually trade big blocks and do not

want to expose their intents to others. Goldman Sachs' *Sigma X* is one such dark pool with big volumes.

Figure 2 from [1] illustrates the modern day stock market structure.

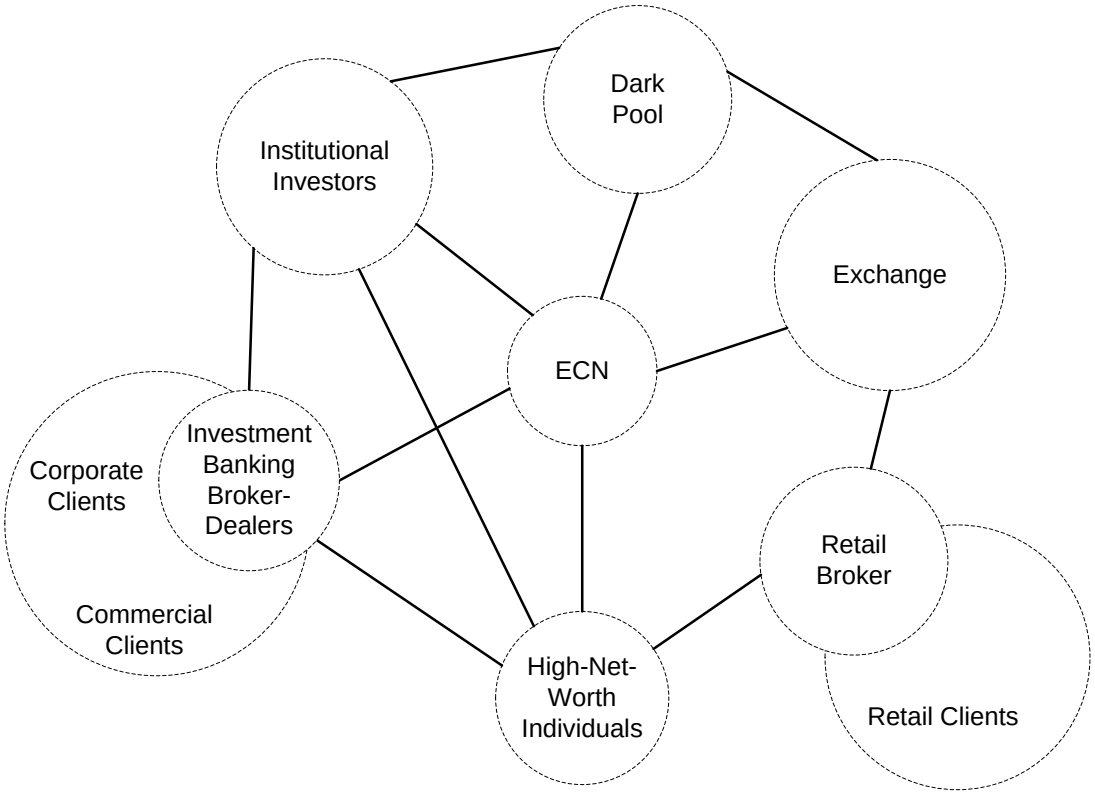


Figure 2: Modern Day Stock Market Structure

The core of modern day stock market structure is the interconnected markets: Exchange, ECN and Dark Pool triangle. Clients can connect to any one of them and are exposed to the whole market. Clients can directly connect to the market or indirectly through brokers or dealers.

The interconnected markets guarantee that the best price is executed for every trade all the time. If one market receives an order to buy and it does not have the best offer, it routes the order out to one that does. The market is liable for the difference if an order is executed at worse price at its venue. A *Smart Order Router* is a computer

algorithm that specializes in splitting the orders and routing them to different venues with the best quotes at the same time. Each market is assigned with a one-character M*arket Participant ID (MPID)*. Table 1 lists some major markets in US.

| Market | Short Form | MPID |
|---|---|---|
| American Stock Exchange | AMEX | A |
| Boston Stock Exchange (Acquired by NASDAQ) | BX | B |
| National Stock Exchange (Acquired by CBOE) | NSX | C |
| FINRA ADF (Quote Display & Trade Report, no Execution) | FNRA | D |
| International Securities Exchange | ISE | I |
| Direct Edge A | EDGA | J |
| Direct Edge X | EDGX | K |
| Chicago Stock Exchange | CHX | M |
| New York Exchange | NYSE | N |
| NASDAQ Stock Exchange (INET) | INET | T |
| Archipelago ECN (Acquired by NYSE) | ARCA | P |
| CBOE Stock Exchange | CBOE | W |
| NASDAQ PSX Stock Exchange | PSX | X |
| BATS Y Exchange | BYX | Y |
| BATS Z Exchange | BZX | Z |

Table 1: Major Stock Markets in US and their Participant IDs

Multiple interconnected markets also present the challenge to disseminate market data information. The traditional market as NYSE receives the orders, matches the trades and publish the trades all by itself. Now there is more information than just trades

to publish and each market holds a partition of the complete market data information. At the same time, a trade can be facilitated between buyers and sellers directly outside all the markets. To report these trades is a challenge too.

It is a lot more complex with regards to quotes. For a stock *symbol*, each market has many bids or asks. All the *bids* are arranged in the descending order while *asks* in the ascending order in a *book* where the best bid and offer (*BBO*), the highest bid and lowest ask are on the top page of the book. The book is dynamic as bids and asks come and go constantly.

There are five level of information in modern stock market to be reported. The higher the level, the more information there is.

| Level | Information | Description |
|---|---|---|
| 1 | Trade | The result of execution of orders from all markets as well as off-market transactions. |
| 2 | NBBO | National Best Bid and Offer. The highest bid and lowest ask across all markets |
| 3 | Top of Book | The best bid and ask on each market |
| 4 | Aggregate Book | Every bid in descending order, ask in ascending order and the aggregate size on them for each market |
| 5 | Order Book | Every limit order and execution of both limit and market on each market. |

Table 2: 5 Levels of Information in Modern Stock Market

The next sections show examples of information of these levels.

**Trade**

Time & Sales ORCL

| Exchange | Price | Size | Time | Date |
|---|---|---|---|---|
| FNRA | 32.37 | 100 | 15:46:11.344 | 03/21/13 |
| FNRA | 32.37 | 100 | 15:46:11.324 | 03/21/13 |
| FNRA | 32.37 | 100 | 15:46:11.304 | 03/21/13 |
| EDGX | 32.37 | 200 | 15:46:11.162 | 03/21/13 |
| EDGX | 32.37 | 200 | 15:46:11.161 | 03/21/13 |
| EDGX | 32.37 | 600 | 15:46:11.160 | 03/21/13 |
| EDGX | 32.37 | 200 | 15:46:11.159 | 03/21/13 |
| FNRA | 32.37 | 100 | 15:46:11.157 | 03/21/13 |
| FNRA | 32.37 | 100 | 15:46:11.156 | 03/21/13 |
| FNRA | 32.37 | 100 | 15:46:11.129 | 03/21/13 |
| FNRA | 32.37 | 100 | 15:46:11.129 | 03/21/13 |
| FNRA | 32.37 | 100 | 15:46:11.113 | 03/21/13 |
| PSX | 32.37 | 100 | 15:46:11.097 | 03/21/13 |
| FNRA | 32.37 | 100 | 15:46:10.962 | 03/21/13 |
| FNRA | 32.37 | 1000 | 15:46:10.953 | 03/21/13 |
| EDGA | 32.37 | 100 | 15:46:10.944 | 03/21/13 |
| EDGA | 32.37 | 200 | 15:46:10.943 | 03/21/13 |
| PSX | 32.37 | 300 | 15:46:10.878 | 03/21/13 |
| FNRA | 32.37 | 100 | 15:46:10.866 | 03/21/13 |
| BYX | 32.37 | 200 | 15:46:10.863 | 03/21/13 |
| BYX | 32.37 | 100 | 15:46:10.837 | 03/21/13 |

Figure 3: Trades

Figure 3 shows some trades from different markets. Trades are also called *prints* or *tape* as they are originally printed on the paper tape. Based on the exchange the stock is listed on, there are three tapes in US stock market:

- Tape A: trades of all stocks listed on NYSE
- Tape B: trades of all stocks listed on AMEX
- Tape C: trades of all stocks listed on NASDAQ

Trades from Tape A and B are reported to a separate organization: Consolidated Tape Association (CTA) from each market real-time. Tape C trades are reported to NASDAQ *Unlisted Trading Privileges (UTP)*. CTA and UTP consolidate all the trades

and report to their clients. Off-market trades are reported to CTA or UTP through *Trade Reporting Facility* (*TRF*) on NYSE or NASDAQ.

**NBBO**

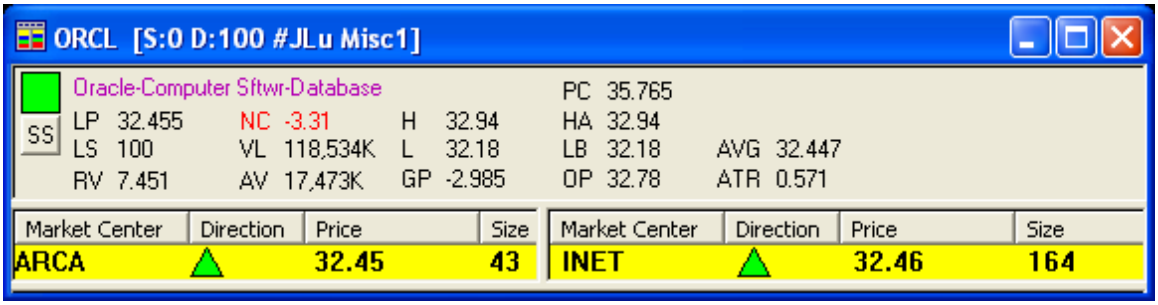


Figure 4: NBBO

Figure 4 shows the current national best bid is from ARCA and national best offer is from INET. Each market reports to CTA and UTP the best bid and offer on their order book, or top of book. CTA and UTP consolidate them, report the highest bid and lowest ask among all markets and mark them as national best bid offer (NBBO).

**Top of Book**

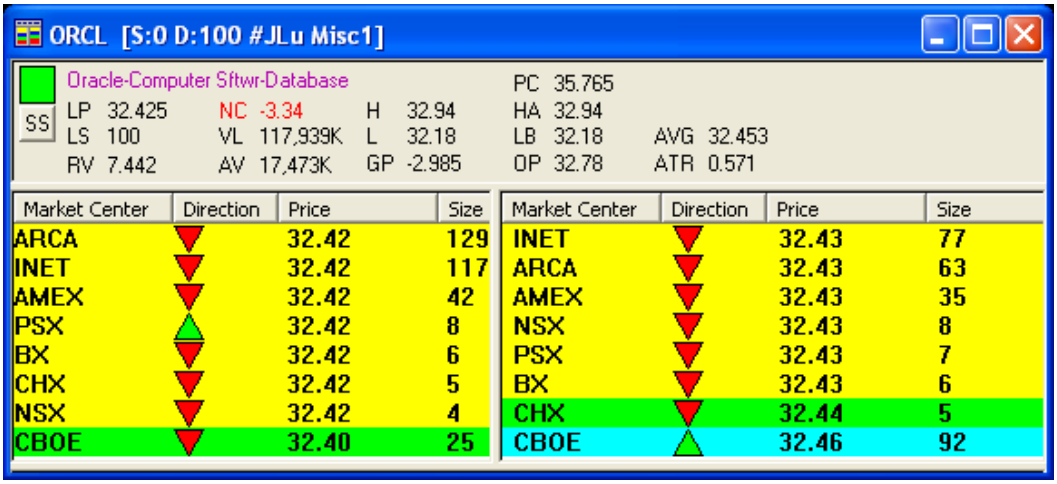


Figure 5: Top of Book

Similar to NBBO, CTA and UTP receive best bid and offer from each market and report all of them under the code of each market. Figure 5 shows the current top of book from each market.

**Aggregate Book**

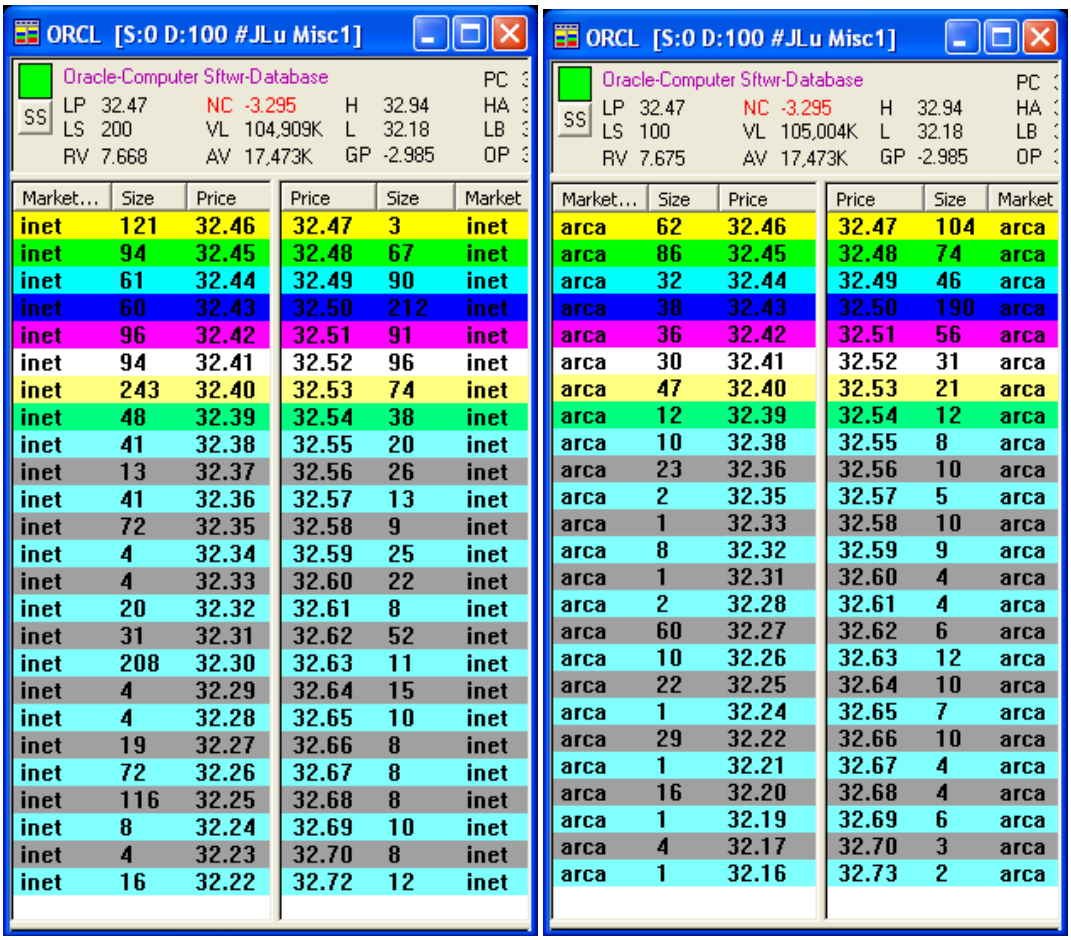


Figure 6: Aggregate Book

Figure 6 shows the aggregate book of *INET* and *ARCA* with bids on left side and asks on right side. Currently NYSE provides market data feed for direct aggregate book.

Others require applications to receive the order book and generate aggregate book internally.

**Order Book**

Order Book provides the information of every limit order: buy, sell, modify and cancel, as well as execution of all orders. Market order can be deduced from order match. Some special orders such as *iceberg order* may only show a small portion of the real size (tip of the iceberg) of the order to protect the trader. Order by Order information offers computer models the opportunity to analyze the market in great detail. Many market making models depend on order by order information.

**DATA FLOW OF US STOCK MARKET**

Figure 7 illustrates the major part of US stock market. Each market, such as *Market A*, contains two parts: *Order Match Engine*, which receives orders from external *Trading Applications* and matches them to generate trades. The orders and trades feed into its *Market Data Engine*. *Market Data Engine* receives orders and trades, generates *Top of Book* and *Order Book*. It may also arrange raw orders into *Aggregate Books*. *Market Data Engine* then reports its trades and *Top of Book* to *CTA* for tape A and B symbols and *UTP* for tape C symbols. Off-market trades are reported through either *NYSE TRF* or *NASDAQ TRF* and *TRF* reports to *CTA* or *UTP* accordingly. *CTA* and *UTP* consolidate trades and quotes and generate *Trade*, *NBBO* and *Top of Book* information for dissemination to *Trading Applications*. *Market Data Engine* also publishes detailed *Order Book* or *Aggregate Book* information to *Trade Applications*.

While *Market Data Engine* of each market is independent from each other, all *Order Match Engines* are interconnected. When *Order Match Engine* of one market does not have the best price to fill the incoming order, it routes out the order to another *Order Match Engine.* The process continues until it reaches one that does.

*Trade Application* is an external application that receives market data and generates orders either manually or automatically. It may only connect to one *Order Match Engine* for all the orders which may be routed out to others for best price. It may also connect to many *Order Match Engines* and decide to send to some based on *Top of Book* of those markets for faster execution.

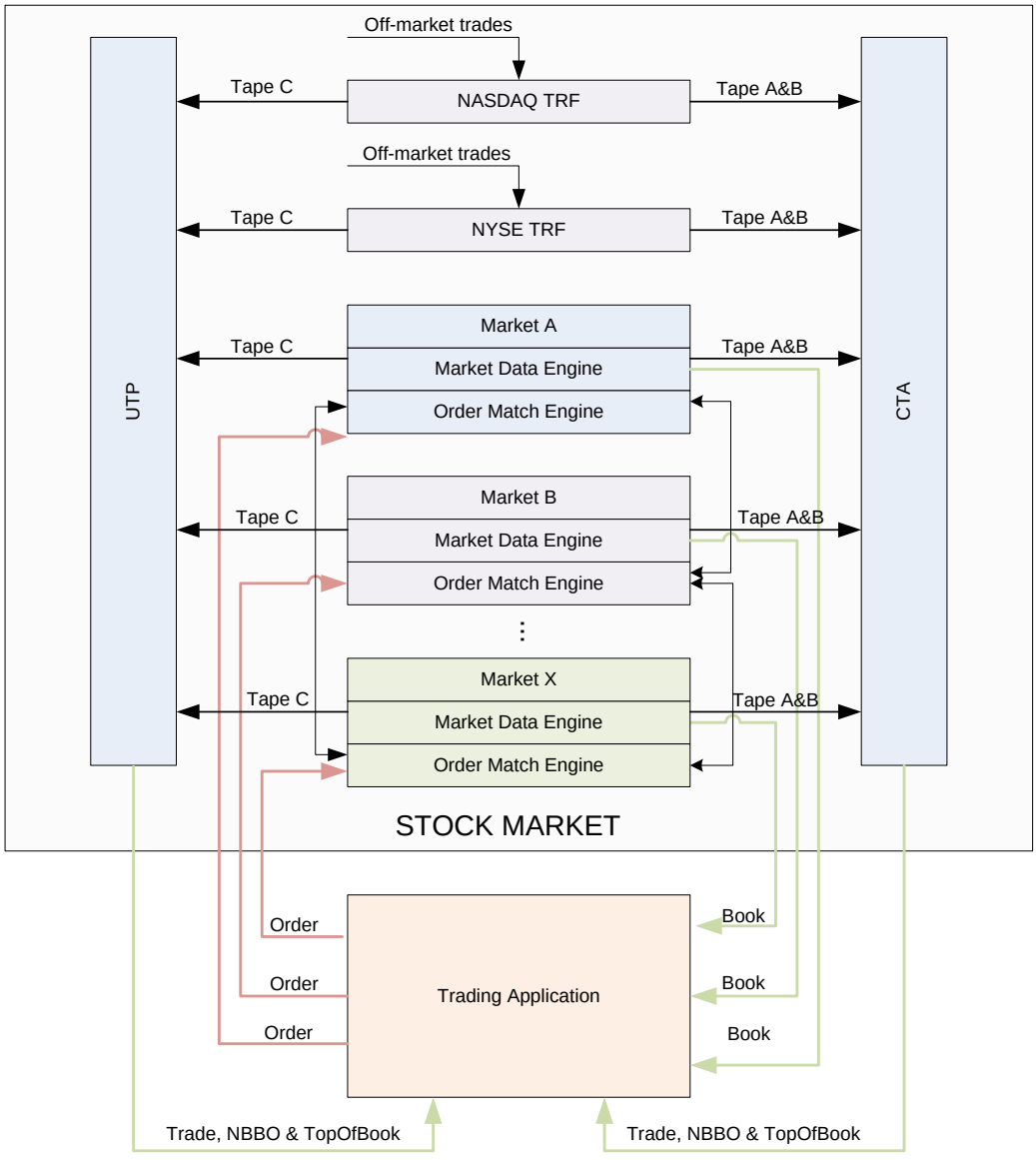


Figure 7: US Stock Market Data Flow

# Chapter 2: Quantitative Model in QUARTS

Before analysis of quantitative model, we begin with a typical day of a trader:

On April 22nd 2013, trader Jason sat in front of a trade station. He browsed quickly through a set of stocks for a familiar pattern: On 5-minute chart, 15-period exponential moving average (15EMA) crosses 50-period simple moving average (50SMA) and stock price forms a higher low. At 10:50 EDT he found it on AAPL (Apple Inc.) and placed an order to buy it on market. The order is filled with 100 shares quickly at $394.95 and the position is opened. He set up 1% target and placed a limit sell order for $399 and a stop order for $392. While continuing to look for more opportunities, he monitors the profit and loss of the open positions. At 14:41 on the day, AAPL reached his target price and the position is closed for profit.

This is a familiar scene of a manual trader. Quantitative model, an abstract representation of a financial decision-making, works no different except that the trader is replaced by a piece of computer software. Figure 8 illustrates the interactions between a quantitative model and its environment. A quantitative model receives *market data* from *Market Data Engine*, analyzes it, generates an *order* and sends it to *Order Match Engine*. It receives *order ACK/Fill* from *Order Match Engine* to for status of the order.

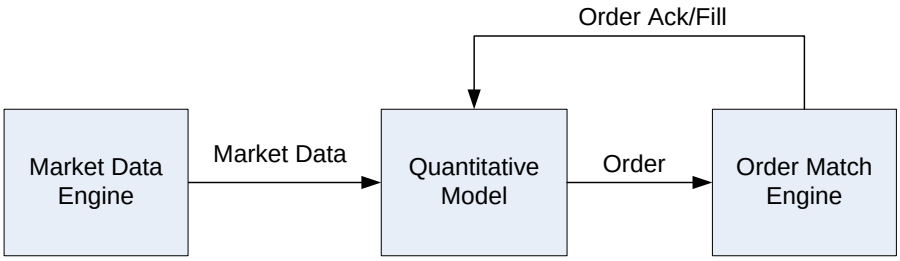


Figure 8: Quantitative Model

### MODEL RUNNING MODE

When some ideas about the market turns into a quantitative model, the quant always wants to test it thoroughly before putting real money into it. The trading without

real money is called *paper trading*. Similarly the account is called *paper account* and virtual money involved is called *paper money*. For a model to be tested, it requires an input from *Market Data Engine* and an *Order Match Engine* to receive orders and to respond with order acknowledgement and order fill messages.

| Order Match Engine / Market Data Engine | Mock (with fake fill & paper money) | Live |
|---|---|---|
| Mock (with historical data) | Back-test | N/A |
| Live | Forward-test | Live |

Table 3: Different Model Running Modes

As we can see in Table 3, based on the mock or live of market data engine and order match engine, there are three modes to run the same model:

- *Back-test***.** The mock version of *Market Data Engine* feeds the model with historical market data recorded from the market. On the other side, the mock version of *Order Match Engine* accepts the order from the model and fills it according to the historical market data at then market conditions and feeds back the order fill message when necessary. A model usually goes through back test mode many times with optimizations or even redesign. And most of the models do not get out of back-test cycle at all.
- *Forward-test***.** The live *Market Data Engine* feeds the model with real-time data. The model acts as if it is running live. However the mock version of *Order Match Engine* determines that only paper money is involved. This is the last necessary step before throwing real money into it. In stock market, history repeats itself, but never exactly. Forward-test discovers whether the model that fits the history well still holds fairly in current market conditions.

- *Live***.** Putting model into live trading is the ultimate goal of a quant. A live model may still go through certain adjustments although mostly operational. One major adjustment is the allocation of capital. Scaling up or down a live model is typical based on market conditions. Shutting it down and bringing it to back-test mode for further research is also possible when it does not perform as expected in the market.

### LIFECYCLE OF A MODEL

In QUARTS, the same quantitative model goes through certain stages. Figure 9 illustrates the lifecycle of a model. As described in introduction, the model is written in Python language. The back-test, forward-test and live trading are C++ applications.

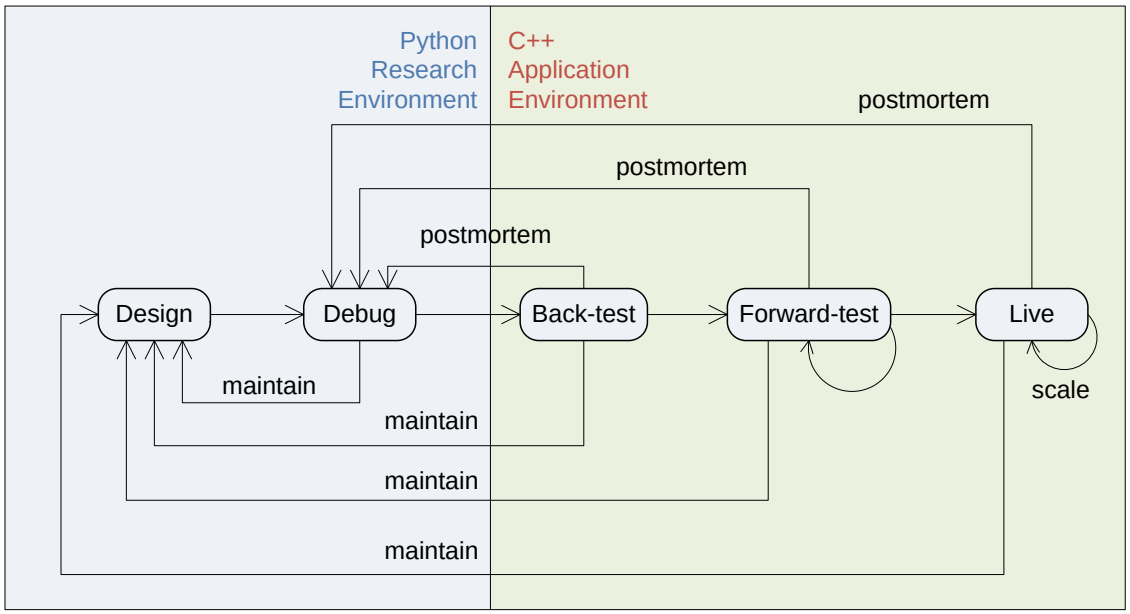


Figure 9: Lifecycle of a Model

- *Design*. Start with ideas and implement the model in Python
- *Debug*. Run model in Python environment for limited range of historical market data

- *Back-test*. Run model in C++ environment for thorough test with full range of historical market data.
- *Forward-test*. Run model in C++ environment with live market data but paper money
- *Live*. Run model in C++ environment with live market data and real money

The usual lifecycle is from design, debug to back-test, forward-test and live. Only models that perform well in the previous stage go into the next one. In back-test, forward-test and live stage, if on any specific day the model is suspected not to run as expected, a postmortem is performed by loading market data of that day into debug environment and debug the model closely. If the model does not perform well in back-test, forward-test even in live mode after a while, it is brought back to design stage for maintenance.

**MODEL OPTIMIZATION**

A well-designed model supports a set of options and parameters so that the model can be optimized through them. Options are configurations that all models share while parameters are model specific. Table 4 lists one common option that can be used by all models.

| Options | Type | Description |
|---|---|---|
| EntryMode | Integer | On signal to take position for symbol<br>0=SingleEntry. No if it had position for the day<br>1=SingleFailure. No if it traded with loss for the day<br>2=SinglePosition: No if it has position currently<br>3=TakePosition: Yes |

Table 4: Options of a Model

Table 5 lists parameters for a model with simple trail stop and take profit exits.

| Parameters | Type | Description |
|---|---|---|
| Trail | Float | Offset in cents for model with trail stops |
| TrailPercent | Float | Offset in percentage for model with trail stops |
| Profit | Float | Offset in cents for model that limits target profit |
| ProfitPercent | Float | Offset in percentage for model that limits target profit |

Table 5: Parameters for a Specific Model

Figure 3 illustrate the optimization process for a simple model that supports just one option and four parameters.

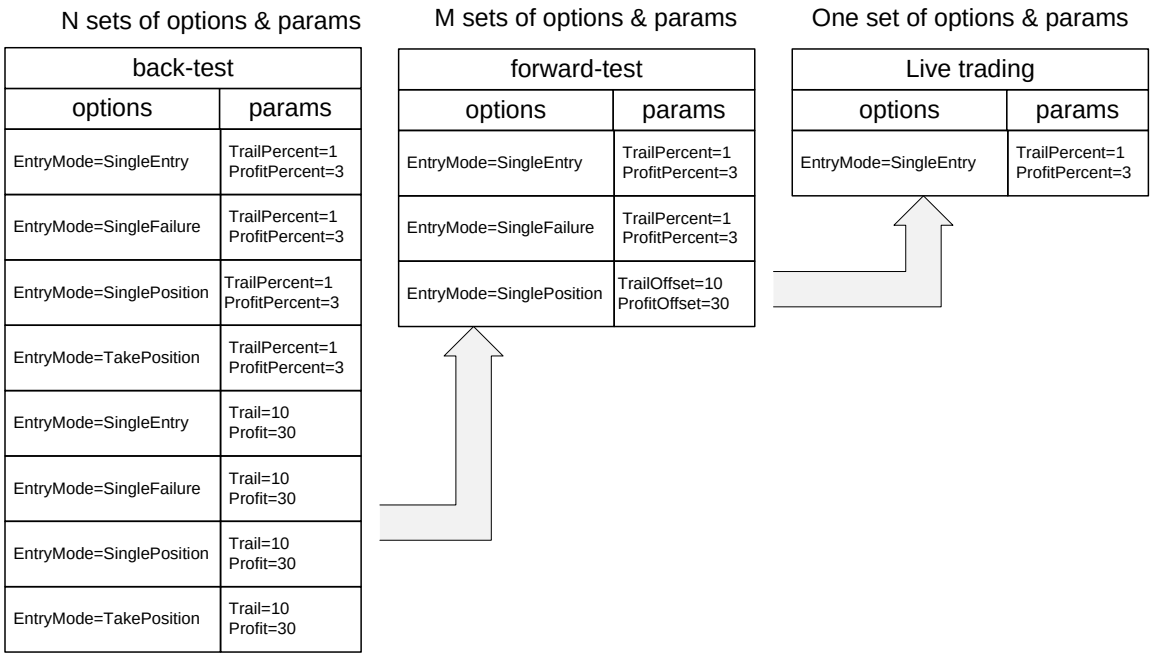


Figure 10: Optimizing a Model

After the model is designed and validated, the usual process is to find the best set of options and parameters that yields optimal results. During back-test, a lot of possible options and parameters are set up for the model to run against historical data. A subset of those options and parameters that have optimal results are set for forward-test. Only one set of options and parameters will be used in live trading if the model survives forward-test at all. Figure 3 illustrates the optimization process driven by option and parameters. The model is back-test with N sets of options and parameters of which only M (<=N) sets of options and parameters are used for forward-test. Finally one set of options and parameters are used for live trading.

## MODEL COMPONENTS

Having analyzed how models interact with their environments in previous sections, this section focuses on the analysis inside a model to determine what a model should be composed of.

The ingredients of trading as Tharp, Van K. analyzes in his book, *Trade your way to Financial Freedom* [2], is illustrated in Figure 11 on the next page with the area indicating the weight of each ingredient.

*System*, the conditions to get in and out of a trade, is the core of any successful trading, to some all of it. However important it is, *System* plays a small part of trading success according to Tharp. *Money Management*, the strategy to allocate limited capital to same or different stocks and to manage risks once in position, is a more important factor to succeed. With a winning system and money management tactic, the trader's success is ultimately determined by *Psychology* factor. The psychology factor determines whether, under any market circumstances, the trader follows his system and money management rules. The weakness of human emotions such as fear and greed often brings

even the greatest traders of the time to their knees by breaking their own rules in difficult market situation [3].

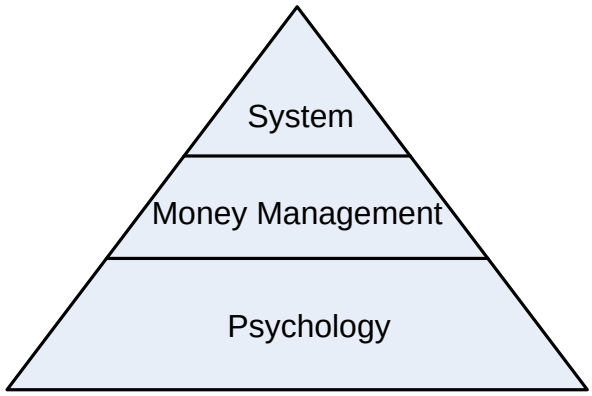


Figure 11: Ingredients of Trading

It is tempting to treat the quantitative model as a whole and to give the quants the most flexibility. And this is the mode of all quantitative model platforms I have researched. However, proper decomposition of the model has potential important benefits without limiting the potential of the model. Software architecture design principles tell us to separate components of the system that change at different speeds for better maintainability. Exit strategies of the model change a lot slower than the entry strategies. In fact, there are only limited a few exit strategies commonly used and trail stop exit is the most common of all. On the other hand, various models are usually characterized by its entry strategies so entry strategies are as varied as models. Different components of the model also help quants to focus on specific areas one at a time. One strategy focuses on generating entry signals, the other one can focus on how big the position should be at circumstances on entry signals.

Among three ingredients to make a good trading system in Figure 11, a quantitative model does not have *psychology* ingredient as it always follows the *system* and *money management* rules set up for it. A *system* should contain both *entry* strategy and *exit* strategy. *Money management* manages capital allocation right before getting into

the position and monitors positions once in it and may trigger exits on risk parameters. The former is called *position sizing* and the latter *risk management*.

While *entry*, e*xit* and *risk management* may trigger orders to buy or sell certain shares of a stock with (limit order) or without (market order) price, a separate component may be necessary to best execute the order. *Execution strategy* can decide, for example, that to buy AAPL, the best way is to place it on INET market at the moment. *Execution strategy* can also place a few smaller orders on different markets to fulfill a big order fast. *Execution strategy* can also place special orders as hidden or iceberg if necessary. Figure 12 illustrates the components of the quantitative model.

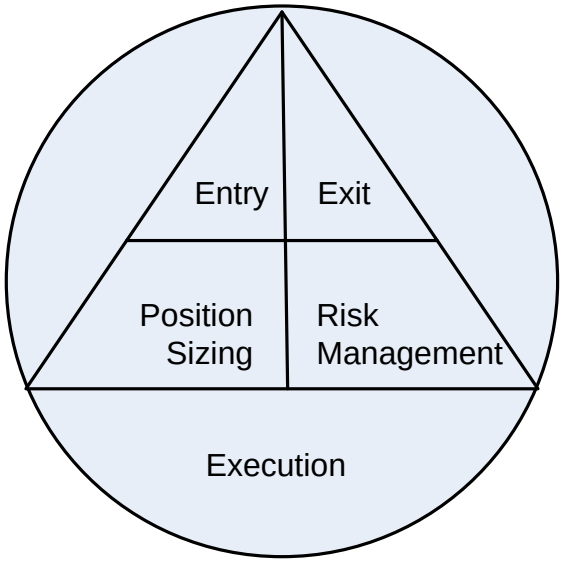


Figure 12: Components of Quantitative Model

- *Entry Strategy*, to open a position of a specific symbol when certain conditions are met.
- *Exit Strategy*, to close the existing position of a specific symbol when certain conditions are met. The exit can be to take profit when target is reached, or to take loss when stop conditions are triggered.

- *Position Sizing Strategy*, to answer "how many shares" question when entry strategy signals.
- *Risk Management Strategy*, to monitors all the positions and to generate exit signals when certain risk conditions are met. These exit signals are the extra exit signals from exit strategy in that these exits are used for special purposes. A simple risk management can be to control the overall daily loss. For example, if daily loss reaches $3000, liquidate all the positions and stop for the day as it could be the market on that day, with unknown reasons, does not fit the model well.
- *Execution Strategy*, to decide the best way to execute an order generated by other components. It determines the destination, the special order instructions, the breakup of orders based on the market conditions. It may also convert market order to limit order with lower prices to avoid potential market order traps when market moves fast and when liquidity dries. Execution Strategy is optional as most models may not need it.

A complete quantitative model depends on five individual strategies. *Entry strategy* relies on *position sizing strategy* to get position size when an entry signal is triggered. *Entry, exit and risk management strategies* depend on *execution strategy* to place orders.

What happens with an order that is only partially filled? If a partial fill happens on the order to open position, the position is a bit smaller than the position sizing strategy hopes. This does not prevent *exit strategy and risk management strategy* from working. Although the order to close position is usually market order that guarantees execution, it is possible some market orders are modified by *execution strategy* to be limit orders.

When the order to close position gets pending and the condition that causes *exit strategy* or *risk management strategy* to place the order still exists, the pending order gets cancelled and a new order to close the rest of position is issued usually at a worse price.

Figure 13 redraws the components of a quantitative model to show dependencies.

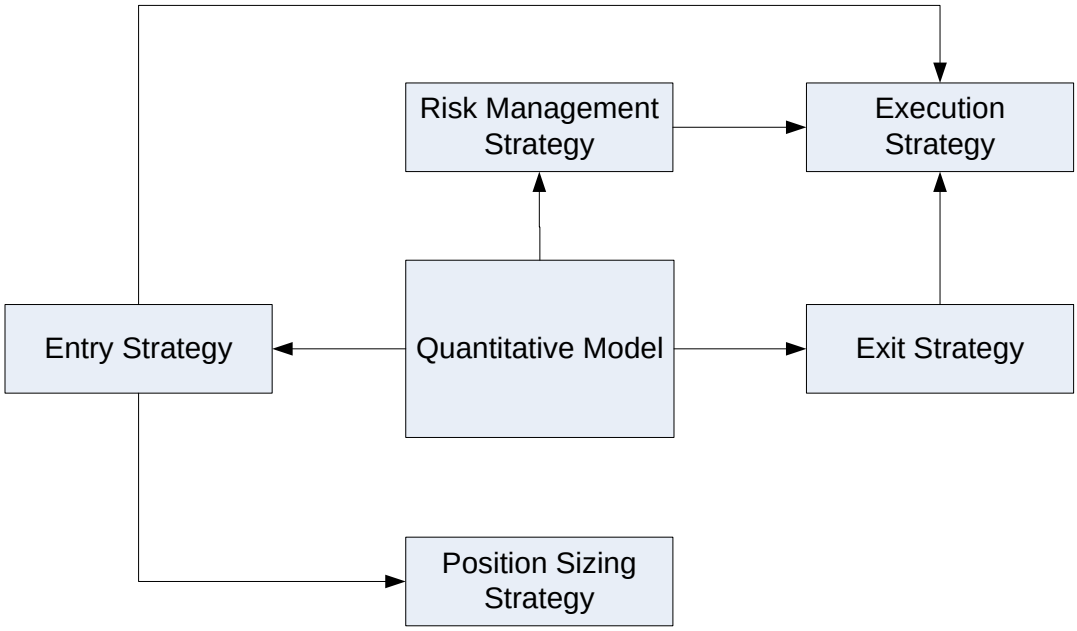


Figure 13: Dependencies of Quantitative Model Components

A model is first triggered by its *Entry Strategy*. When conditions are met, *Entry Strategy* gets entry direction (long or short), symbol (that triggers the signal) and price (current price of the symbol). It consults *position sizing strategy* for the entry size of the symbol based on current capital, position and price of the symbol. It then asks *execution strategy* to place the order. After a position is initiated, *exit strategy* becomes active to close the position at certain conditions. *Risk management strategy* also becomes active to monitor overall positions. Either *exit strategy* or *risk management strategy* triggers to close positions. *Entry, exit and risk management strategies* eventually generate orders which are sent to *execution strategy* to be executed.

## MODEL DATA FLOW

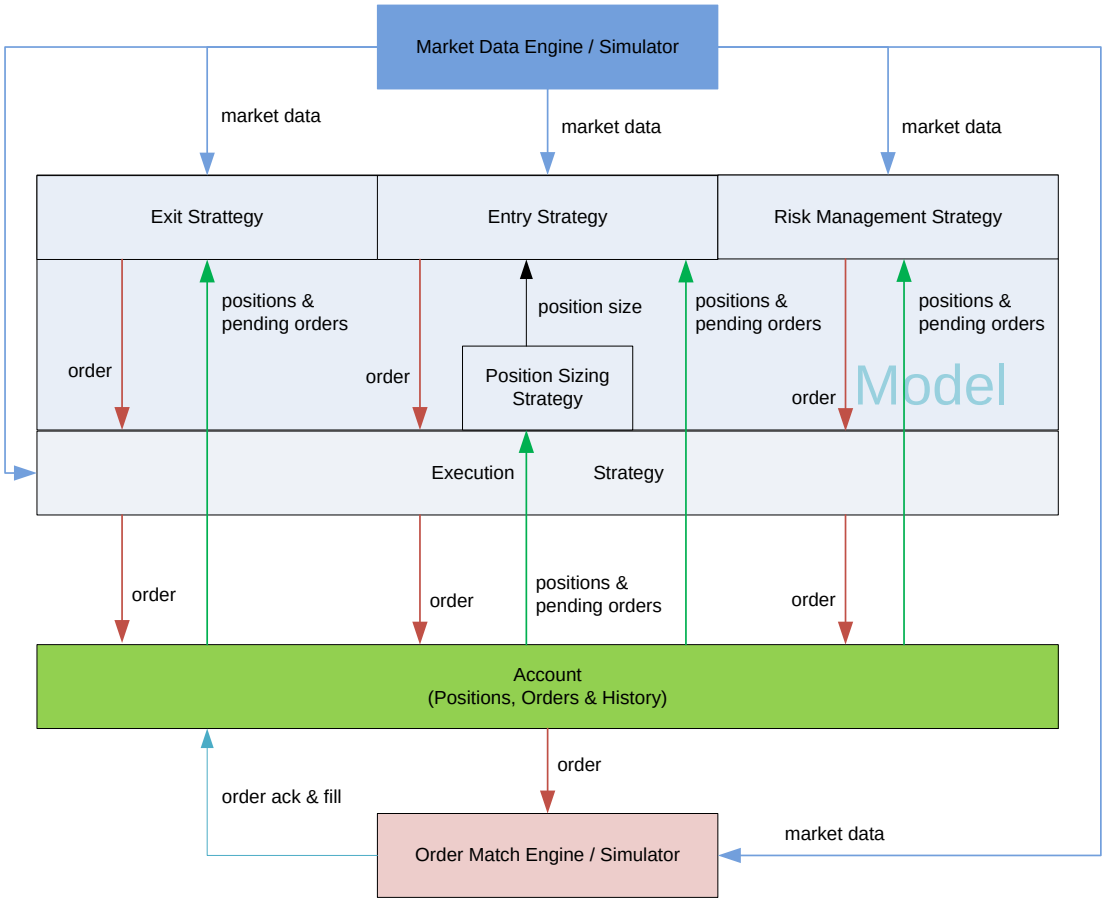


Figure 14: Major Dataflow in Quantitative Models and Environment

Components of the model and components interacting with the model are put together in Figure 14 to illustrate data flows among them. *Account* component is added to manage orders and positions. Each component is described below:

- *Market Data Engine* is the live market data source component from stock market center or its agency.

- *Market Data Engine Simulator* reads recorded market data files in the past and feeds it through the system.
- *Model* comprises five strategy components: *Entry Strategy, Exit Strategy, Position Sizing Strategy, Risk Management Strategy and Execution Strategy*. These strategies can be model specific, or simply instances from established strategy library. Different strategies act similarly in that they all analyze the market data input together with current account information and may generate one or more orders from inside.
- *Execution Strategy* accepts orders from *Entry, Exit* and *Risk Management Strategy*, and redirects it to *Account*.
- *Account* maintains pending orders as well as established positions. It receives order tickets from model, keeps it in its pending orders list and sends the orders out to *Order Match Engine/Simulator*. It expects order acknowledgement and order fill messages from *Order Match Engine/Simulator* to establish the position and update its position list.
- *Order Match Engine* is the live component of the stock market where the orders are executed or its agency.
- *Order Match Engine Simulator* gets market data and simulates the live *Order Match Engine* to execute both market and limit orders.

Figure 14 illustrates data flow as an integration view. It is easier to understand being described in scenarios. In *Entry Strategy* scenario data flows as following:

1. *Market Data Engine / Simulator* feeds the piece of *market data* to *Entry Strategy* of the model.
2. *Entry Strategy*, after analyzing *market data*, decides to generate an order.

3. *Entry Strategy* asks *Position Sizing Strategy* to determine the size of the order.
4. *Entry Strategy* issues the *order* with type (limit/market), price and size to *Execution Strategy* if the size is not zero.
5. *Execution Strategy* receives the order, break it if necessary and decides the target markets to send to. It then sends the modified orders to *Account*
6. *Account* picks up the *order*; put it into its pending orders list and sends to *Order Match Engine / Simulator*.
7. *Order Match Engine* or *Simulator* sends back *order ack* to *Account* and starts order matching process.
8. *Account* updates status of the order accordingly.
9. In simulation mode, *Order Match Engine Simulator* receives *market data* from *Market Data Engine / Simulator* to feed its order matching process.
10. When match is successful either immediately or later on, *order fill* message is sent to *Account* from *Order Match Engine* or *Simulator*.
11. *Account* removes the order from its pending order list and adds position to its position list on receipt of *order fill*.

Data flows in the scenario of *Exit Strategy* and *Risk Management Strategy* differ from *Entry Strategy* in step 3 where the size of the position instead of the size that *Position Sizing Strategy* calculates is used.

# Chapter 3: QUARTS Architecture

Quantitative Research and Trading System (QUARTS) is envisioned to be:

- **A trading system**. A quantitative model is like a fast and active manual trader. The system should be able to manage the model the same way that it manages the trader.
- **A quantitative model trading system**. The system should be able to support quantitative model as Python script, to feed market data and necessary charts and studies and to capture the order out of it to send to the external markets.
- **A quantitative model postmortem system**. After the model runs live for the day, the system should collect enough information to reconstruct the environment, the conditions and the signals it triggers. The system should also be able to rerun the model with historical data of the day in back-test environment and expect it to trigger the same way as is in live market.
- **A quantitative research system**. The system should support quantitative researchers to develop, debug, back test, forward test and optimize the model fairly easily.

The rest of the chapter introduces the design methodology and then applies it to the design of QUARTS architecture.

## DESIGN METHODOLOGY

In *Enabling Iterative Software Architecture Derivation Using Early Non-Functional Property Evaluation* [4], Barber, K.S, Graser, T and others introduced three layers of architectures of a system later called blueprints: *Business Blueprint*, the architecture representing functional requirements in problem domain, *Solution Blueprint,* the architecture representing the solution of the problem defined in *Business Blueprint*

and *Deployment Blueprint,* the architecture representing the physical layout of the components from *Solution Blueprint*.

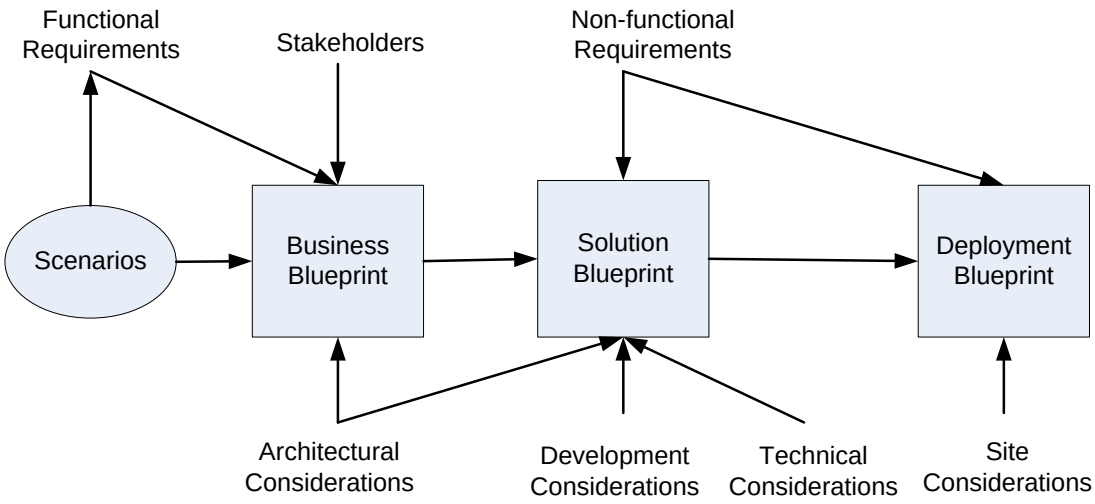


Figure 15 Architecture Derivation Flow

Figure 15 illustrates the flow to derive the system architecture step by step. We start from identifying system stakeholder and major scenarios. From scenarios we collect functional requirements. Based on stakeholders, scenarios and function requirements we bootstrap the first *Business Blueprint* which is then refined based on architectural considerations. Then we collect non-functional requirements (NFR) of the system. With architectural, development and technical considerations, we derive *Solution Blueprint* from *Business Blueprint* and non-functional requirements. Based on the non-functional requirements on the specific site to be deployed, we derive *Deployment Blueprint* by arranging components from *Solution Blueprint*.

More details about software architecture and design methodologies are in Appendix.

### STAKEHOLDERS OF THE SYSTEM

Stakeholders are anyone impacted by the system under development. Some manage the system while others use the system. Some interact with the system constantly while others interact with the system when necessary.

| Role | Description |
|---|---|
| Quant Researcher | Design and debug, back-test, forward test and optimize the model |
| Trader | Analyze both historical and live market data as well as trades positions with paper money or real money manually |
| Quant Trader | Start, stop and scale live quantitative model. Also act as a trader |
| Risk Manager | Oversee both firm wide risk of all trading models and individual models or traders to make sure they are under control |
| Back Office Admin | Handle exceptional issues of the system from inside. Mitigates trade differences between system and clearing house at the end of day |
| System Admin | Configure system and manage both live and test accounts through tools provided by the system |
| Trader Support | Help to verify market data and trade issues |
| Operations Support | Manage hardware and software to run at optimal state with normal situations as well as special situations |
| Network Engineer | Manage and optimize network performance |
| Compliance Officer | Research specific trades after the fact to make sure they are appropriate when being asked by regulators |
| Software Developer | Maintain and optimize the system |

Table 6: Stakeholders of QUARTS

### MAJOR OPERATING SCENARIOS

Major operating scenarios are used to identify major operating scene of stakeholders of the system. The process of scenario identification results in the discovery of major functional requirements of the system. While some scenarios are carried out by only one stakeholder, many others are done through co-operations of stakeholders. Table 7 lists the major scenarios for QUARTS.

| Scenario | Functions |
|---|---|
| Design model | Open Python Environment<br>Design options and parameters (the model supports)<br>Design and implement entry strategy<br>Design and implement exit strategy<br>Design and implement position sizing strategy<br>Design and implement risk management strategy<br>Assemble strategies into model<br>Save model to text file. |
| Debug model | Load model from text file into Python environment<br>Create local model instance (with options and parameters)<br>Create local account<br>Associate local account with local model instance<br>Run local account<br>Analyze account for results |

Table 7: Major Operating Scenarios of QUARTS

| Scenario | Functions |
|---|---|
| Back-test model | Add model to system<br>Generate sets of options and parameters for the model and add to system<br>Create model instances based on sets of options and parameters created<br>Create back-test accounts in system<br>Associate each account with one model instance<br>Schedule accounts for back-test for specified historical period<br>Analyze accounts for results |
| Forward-test model | Select M model instances surviving back-test<br>Create M forward-test accounts in system<br>Associate each of M accounts with respective model instances<br>Schedule M accounts for forward-test<br>Analyze accounts for results at the end of day. |
| Run model live | Select a subset of model instances with satisfactory forward-test results<br>Create live accounts in system with allocated capital<br>Associate accounts with respective model instances<br>Schedule accounts for live trading<br>Scale the account (of the model instance) up or down<br>Start, stop the model instance<br>Liquidate account<br>Analyze each account periodically |
| Maintain model | Disable all accounts associated with the instances of the model<br>Load model file into Python environment<br>Make necessary changes as if in *Design a model* scenario |

Table 7: Major Operating Scenarios of QUARTS (continued)

| Scenario | Functions |
|---|---|
| Perform model postmortem | Disable the account associated with model instance (to be investigated)<br>Generate debug script<br>Load debug script into Python environment<br>Debug script |
| Perform trade | Receive historical market data<br>Analyze historical market data (including charts and studies)<br>Receive current market data<br>Analyze current market data (including trades, quotes and book)<br>Place an order<br>Modify the order<br>Cancel the order<br>Receive order status (including order acknowledgement and order fill)<br>Follow *Analyze account* scenario for current positions |
| Analyze account | Analyze current cash balance<br>Analyze current working orders<br>Analyze order activities (filled orders and cancelled orders) for the day<br>Analyze current positions<br>Analyze cash flow history<br>Analyze order history<br>Analyze trade history<br>Analyze position history<br>Plot order history on chart<br>Plot trade history on chart<br>Plot position history on chart |

Table 7: Major Operating Scenarios of QUARTS (continued)

| Scenario | Functions |
|---|---|
| Manage accounts and models | Load models into to system<br>Create model instances (by creating options and parameters for model)<br>Remove model instance from system<br>Remove model from system<br>Create accounts (back-test, forward-test or live)<br>Enable or disable account<br>Liquidate account<br>Scale account up or down (with capital)<br>Delete account<br>Associate account with model instance<br>Disassociate accounts from model instance |
| Monitor and manage risk | Monitor firm-wide loss of all live accounts<br>Monitor position distribution of all live accounts<br>Export all live positions (for external risk management tools)<br>Monitor loss of every live account (not to exceed daily loss limit)<br>Monitor loss of every position (not to exceed max loss limit)<br>Notify traders for oversized risks<br>Liquidate account |
| Investigate trade issue | Record market data (UTP and CTA feeds into files)<br>Initiate trade issue<br>Verify trade issue<br>Report trade issue<br>Parse market data file (for the symbol into text file.)<br>Locate the trade (in the text file.)<br>Investigate the context around the issue |

Table 7: Major Operating Scenarios of QUARTS (continued)

| Scenario | Functions |
|---|---|
| Break trade | Locate the trade in account<br>Insert a special break operation to undo the trade |
| Investigate market data issue | Verify market data issue<br>Report market data issue<br>Record market data (UTP and CTA feeds to files)<br>Parse market data file (for the symbol into text file)<br>Locate market data issue (based on the timestamp)<br>Investigate trade context (around reported time of specified symbol.)<br>Investigate network context |
| Start of day pre-trade preparation | Load easy to borrow list from brokers for list of short-able symbols<br>Prepare active forward-test and live accounts for the day |
| End of day post-trade operation | Reconcile end of day positions with clearing house<br>Generate order audit trail (OATS) file and submit to FINRA |

Table 7: Major Operating Scenarios of QUARTS (continued)

### BUSINESS BLUEPRINT

The goal of business blueprint is to define the problem accurately. Static views are used to represent functional requirements of the problem domain. As only domain knowledge is required to understand the architecture, it is an effective way to communicate with stakeholders with different backgrounds. On the other hand, it is also a bridge between stakeholders and developers.

### BOOTSTRAP

The initial business blueprint is bootstrapped from stakeholders, scenarios and major functional requirements. Each stakeholder becomes at least one component with functions it performs. Sub-scenarios that are embedded in other scenarios become a separate component to reduce redundancy. Scenarios that are performed by single stakeholder become components with a new role from which the stakeholder inherits. Figure 16 on the next page shows the initial bootstrap of business blueprint in UML [15].

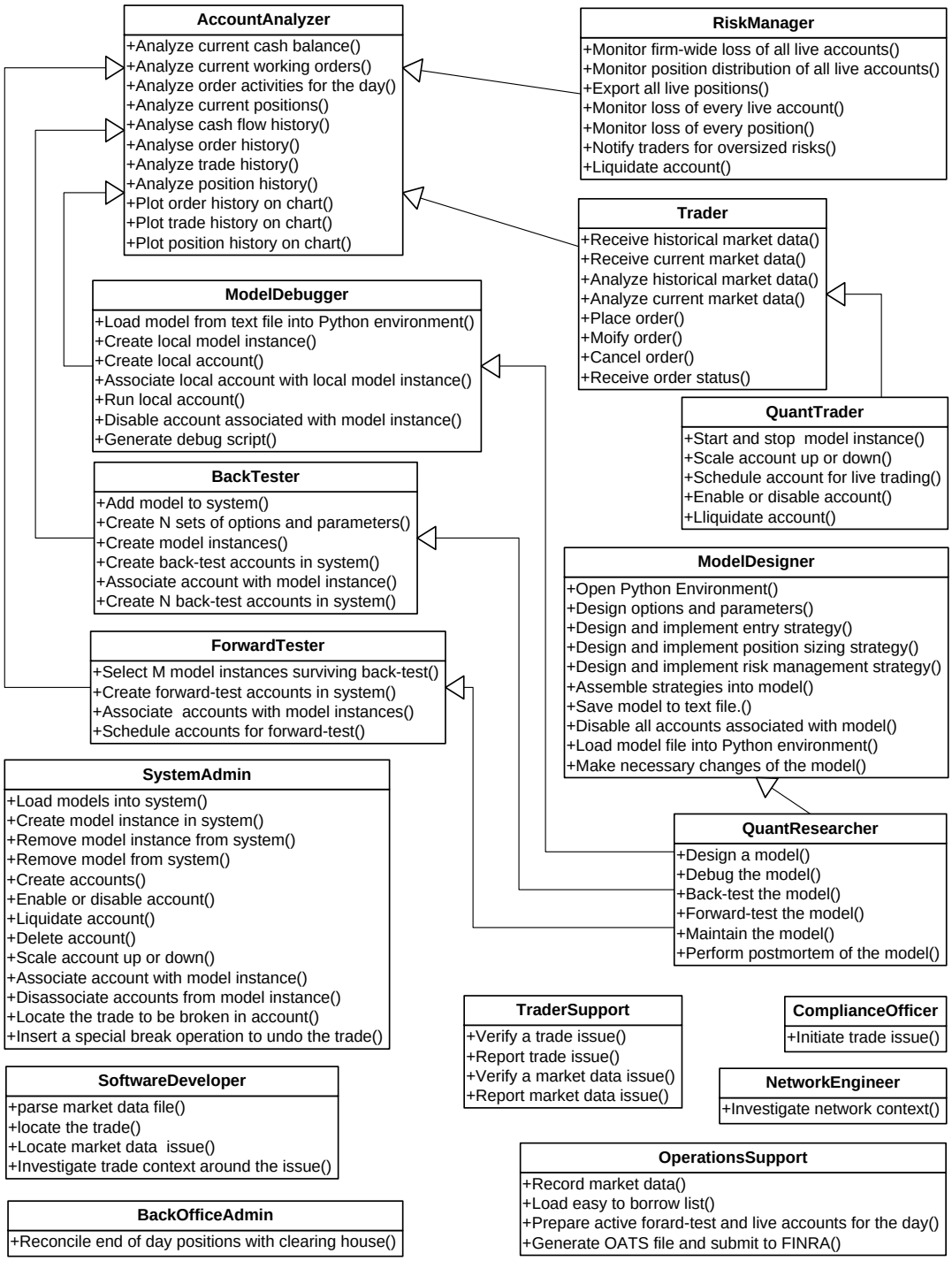


Figure 16: Bootstrapped Business Blueprint

**Refinement**

The initial business blueprint makes sure all major functional requirements are captured accurately. The refinement is to make it architecturally sound. A good architecture should be more cohesive and less coupling. To achieve this, the refinement removes redundancy of services among components. The shared services may be removed from one component and a dependency is created from this component to the other. Or, when the services are shared by multiple components, it is removed from all of them and a new component with these services is created for other components to depend on.

It is clear to see from initial blueprint that Account, Model and Model Instance are three objects many components provide services for. So these three objects become separate components. Figure 17 illustrates the core components

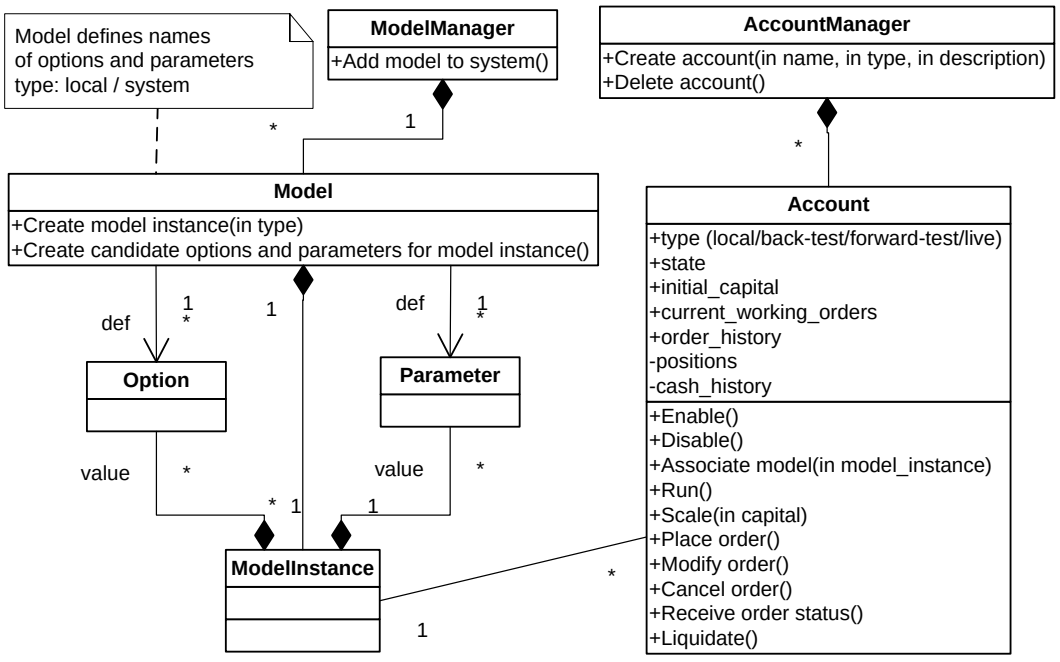


Figure 17: Refined Business Blueprint - Common Components

The rest of business blueprint is illustrated Figure 18. The dependency to core components is implied.

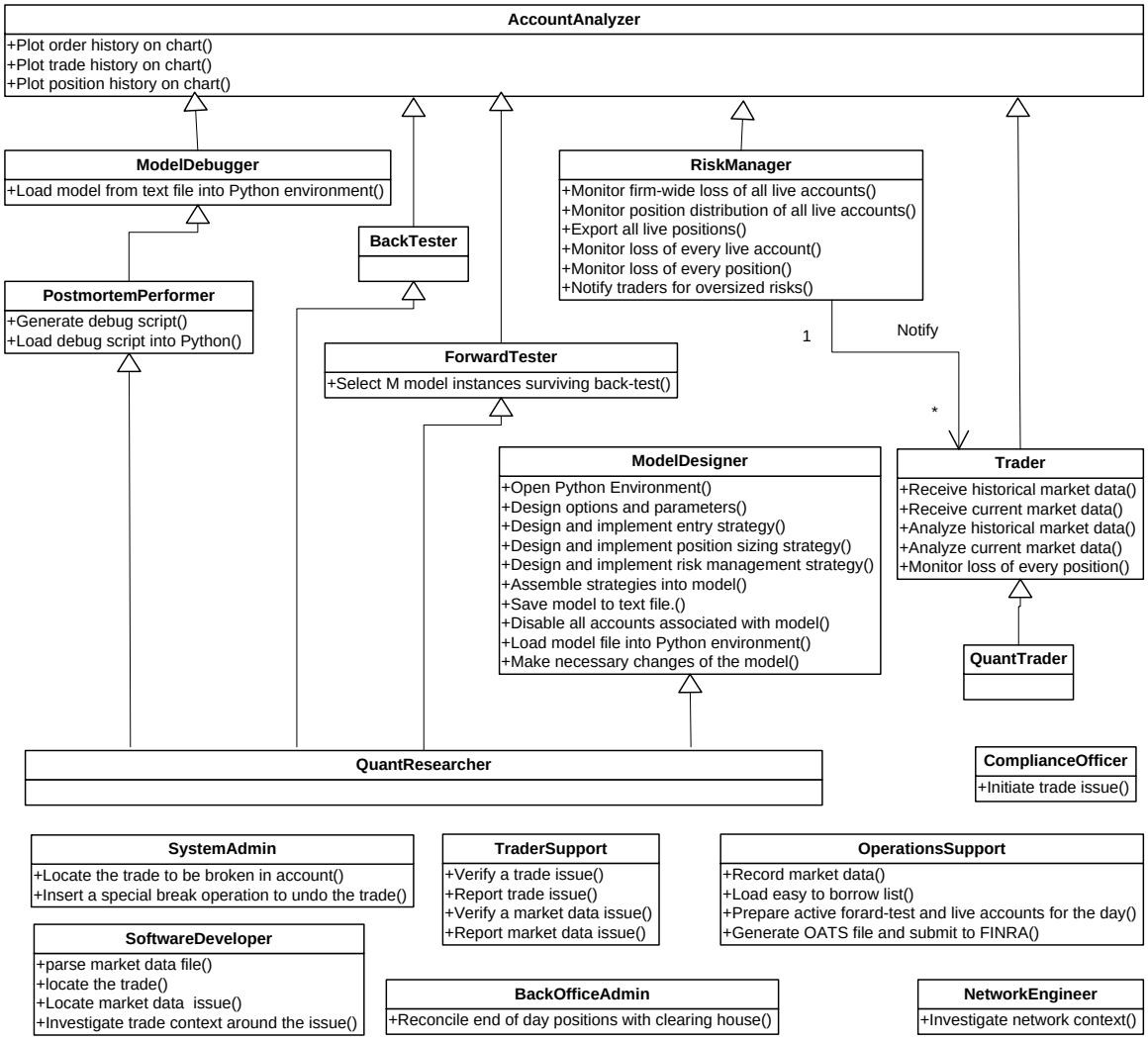


Figure 18: Refined Business Blueprint – Other Components

## SOLUTION BLUEPRINT

Business blueprint presents the problem, which is the functional requirements of the system in a logically easier to understand diagram. Solution blueprint, as its name

implies, presents the solution for the problem in diagram format. The solution involves technical aspects such as database and network. At the same time, solution is required to satisfy non-functional requirements (NFP) of the system too. Unlike functional requirements, NFP is typically implicit from stakeholders. Some of them are discovered by asking stakeholders the right questions while others are discovered directly by architects based on their experiences. One way to acquire NFP is to categorize all possible ones and then to determine whether the system requires them.

**Categorize Non-functional Requirements**

*Software Architecture in Practice* [17] put all non-functional requirements into six categories (USTAMP) listed in Table 8:

| NFR | To Measure |
|---|---|
| Usability | how easy it is for user to accomplish a desired task and the kind of user support the system provides. It includes learning system features, using system efficiently, minimizing impact of errors, adapting system to user needs and increasing confidence and satisfaction. |
| Security | system's ability to resist unauthorized use while still providing its services to legitimate users. |
| Testability | how easy the system can be tested for enough coverage and whether the system provides enough tools to reproduce, analyze and fix the bugs. |
| Availability | how quickly system recovers from failures of certain components. Same as reliability |
| Modifiability | cost of change. Either the change of configuration or the change of source code. |
| Performance | how long it takes the system to respond when an event occurs. |

Table 8: Categories of Non-functional Requirements

And *Modifiability* includes eight sub-categories as illustrated in Table 9.

| Modifiability Sub-category | To Measure |
|---|---|
| Scalability | scale of performance improvement of system with enhanced resources such as number or speed of processors, capacity of RAM, network bandwidth and hard drive speed without code change. |
| Configurability | how easy system can be configured to run with different environments without code change. |
| Supportability | how easy the system can be supported without code change in case of certain issues. |
| Maintainability | how easy to make changes to fix bugs |
| Extensibility | how easy to add new features |
| Flexibility | how easy to change existing features |
| Reusability | how easy components of system can be used in other versions or in other systems. |
| Portability | how easy to move implementation of system to other environments |

Table 9: Sub-categories of Modifiability

**Non-functional Requirements of QUARTS**

Table 10 below list 10 non-functional requirements of QUARTS

| Non-function requirements | Description |
|---|---|
| Performance: low latency for market data | The market data should be able to reach model server within 150 ms or less. It should reach the end user in 300 ms or less. |
| Performance: fast order roundtrip | The order reaches outside market in 300 ms or less for quantitative models.<br>The time when user places an order to when the user receives a visual message of acknowledgement should take 1 second or less. |
| Performance: high throughput market data | The system should have constantly high network throughput to avoid sudden peaks and troughs for all components in system to work smoothly. |
| Security | The system should not expose itself to unauthorized computers and users. All operations from users are saved into audit trail. |
| Availability: staged recovery | The system should be able to recover from single point failure of power, network, computer hardware, database and application for critical components in 1 minute during normal operation times through automatic detection or manual intervention.<br>If hot failover fails, the system should be able to recover manually by restarting the failed component at an alternative location in 10 minutes.<br>Upon unlikely event of complete system crash, the system should have a way to dump all open positions and pending orders. The system should have alternative ways to close positions and cancel pending orders either through a simple tool or by calling the brokerage. |

Table 10: Non-functional Requirements of QUARTS

| Non-function requirements | Description |
|---|---|
| Modifiability-Scalability | By investing more powerful computers, the system should be able to handle more market data without specific bottlenecks. Market data volume doubles every 12-18 months. The system should be able to handle the increased data volume by putting online more powerful computers, RAMs and networks. |
| Modifiability-Extensibility: new market data sources and execution venues | The system should be able to add new market data source components and new execution venue components through configuration without modifying the existing components. |
| Modifiability-Configurability: different network environments | The system should be able to be deployed to different data centers with various network environments with minimal effort. |
| Testability: built-in mechanism to help reproduce bugs | The most difficult part for this kind of system is to reproduce the issue occurred during the day at some point. For one reason some issue cannot be reproduced, as the system cannot place a real order to the market. For another reason that around 40G – 200G bytes of data enters the system and it's hard to locate the packet that causes the issue.<br>The system should have mechanisms to facilitate the ongoing process of bug reproduction, bug scenario narrowing to help fix them quickly. |
| Usability | The system should be able to run within limits without user intervention.<br>The system should be able to have highly customizable user interface to help end users to do daily job efficiently. |

Table 10: Non-functional Requirements of QUARTS (continued)

**Solution Blueprint – Common Components**

The solution to the problem presented in business blueprint for common components is a shared repository architectural pattern where Database Management System (DBMS) plays a central role. Figure 19 illustrates the Entity-Relationship (E-R) Diagram representing details about core tables in QUARTS.

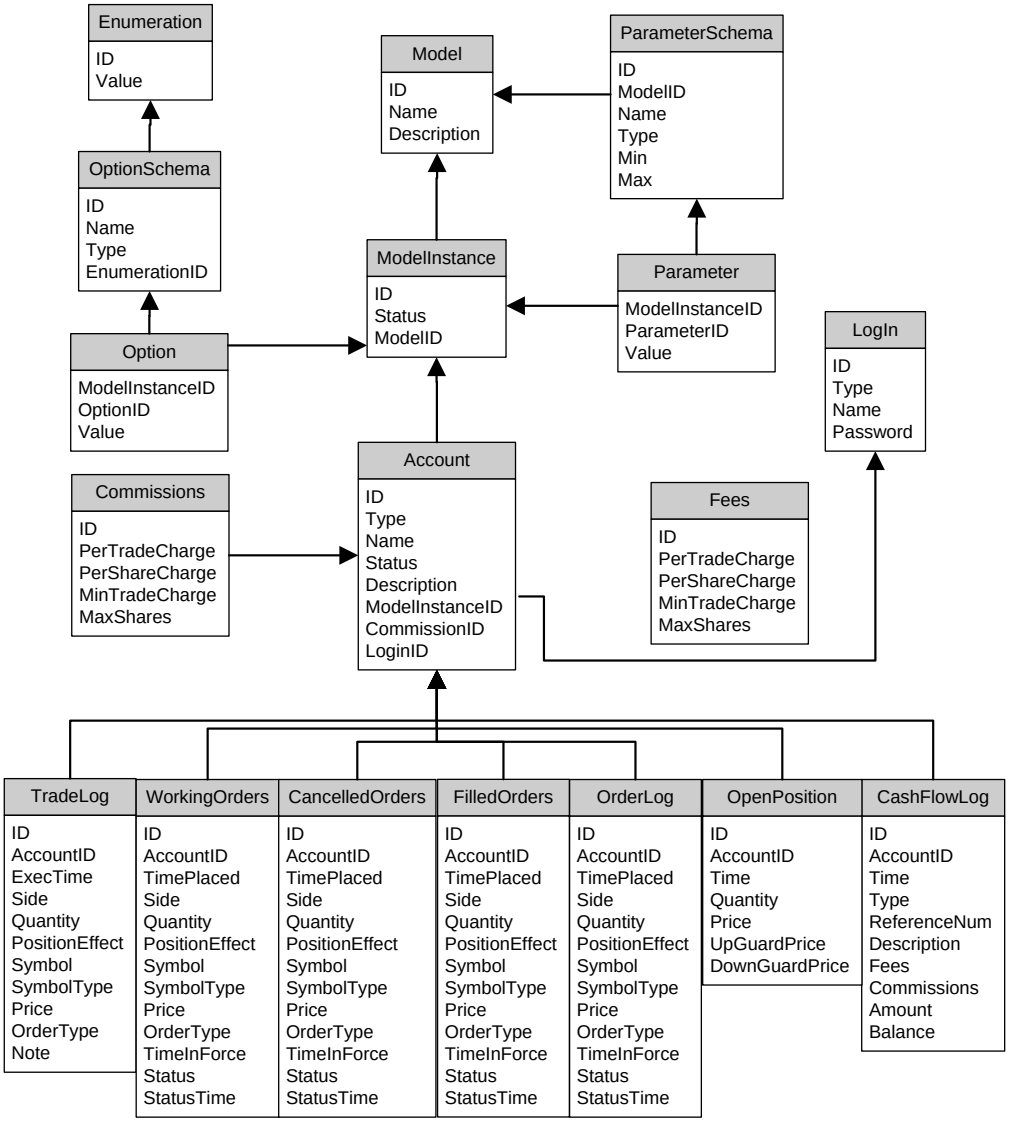


Figure 19: Solution Blueprint Core Components E-R Diagram

**Solution Blueprint – Major Components**

Performance, security and scalability non-functional requirements make the system to follow client-server architectural pattern. The server side components reside in a financial data center close to major markets for low market data latency and fast order flow. The client side components communicate with a security-enforced subset of all server components. Figure 20 on the next page illustrates the major architecture of QUARTS system.

QUARTS is the *Trading Application* illustrated in Figure 7 in Stock Market Structure chapter, which receives real-time market data from *Market Data Engine* and sends orders to *Order Match Engine*. The system also connects to external *Tick Data Service* to receive high quality historical market data.

Most of the client side components are directly from business blueprint. *Local Database* is used on the client side to synchronize with *Master Database* in datacenter overnight to speed up local back-test. *Market Data Proxy* is the component that feeds real-time market data to client components.

Three groups of server components exist on server side: those which handle market data, those which handle execution and those which serve clients directly. Market data group of servers follow pipe-and-filter paradigm where *Feed Handler* receives raw market data in different formats, transform it into internal format with different organization and put it to *High Volume Market Data Bus* which is a set of multicast groups on high speed data center network of 1-10GBPS. *Historical Market Data Server* receives market data from *High Volume Market Data Bus* and transforms it into different charts. Market Data Server follows publisher-subscriber architectural style to filter to *Low Volume Market Data Bus* only market data requested by client side components through *Market Data Proxy*.

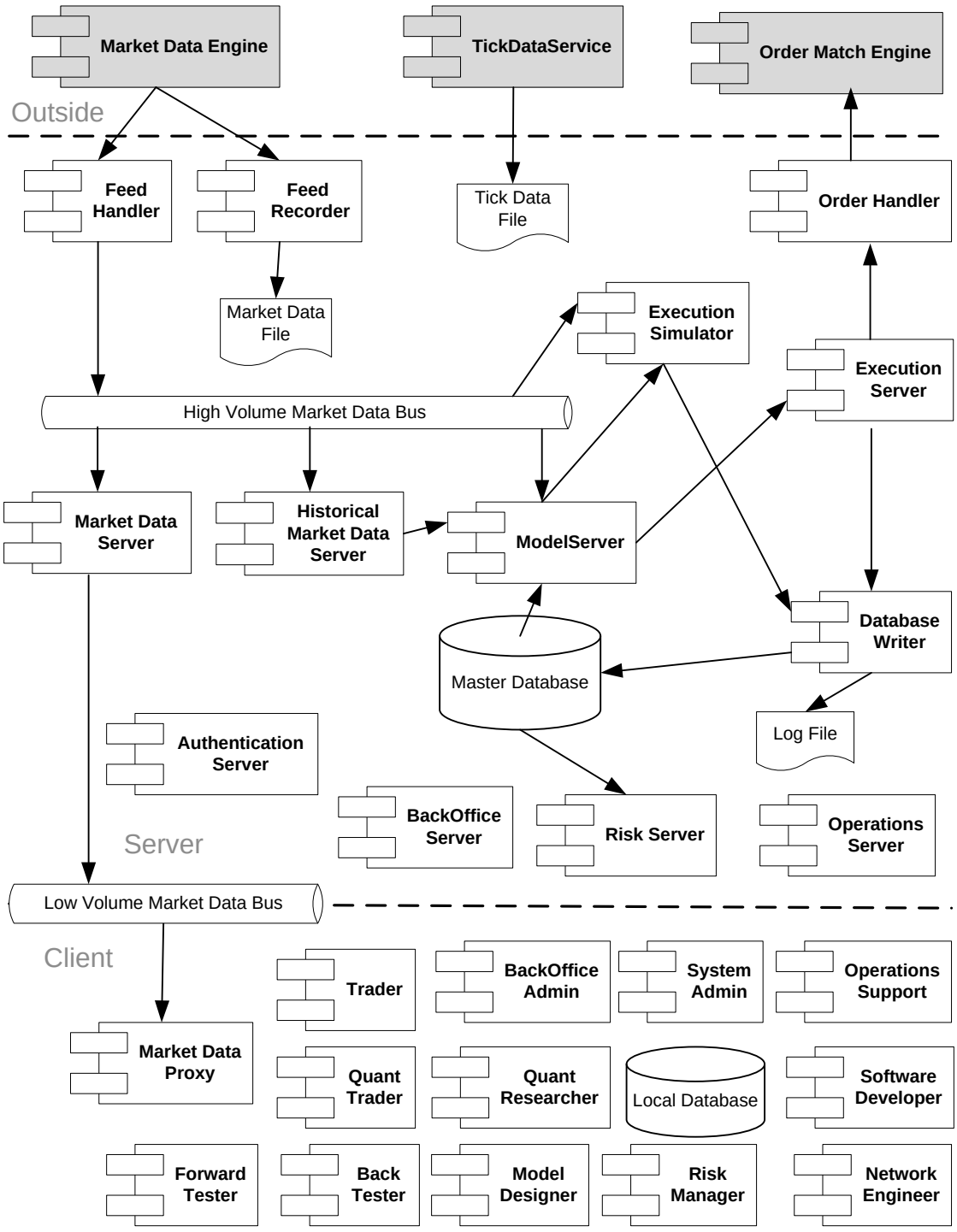


Figure 20: Solution Blueprint – Major Components

Quantitative models are loaded from *Master Database* for forward-test and live trading purposes. Hosted in Model Server, they consume real-time market data off High Volume Market Data Bus and charts from Historical *Market Data Server*, send orders to *Execution Server* or *Execution Simulator* based on the mode of model. *Execution Server* and *Execution Simulator* hold all the information about the accounts that are associated with models and write updated account information such as orders and positions to *Master Database* through *Database Writer. Database Writer* is the component to isolate database from other components for performance and availability. Database I/O can be slow or even down for certain reasons and cannot be depended on real-time. *Database Writer* caches all the database IO requests, writes them to *Log File* as well as *Master Database*. For any database issues, Log File acts as a file form backup. Order Handler sits between *Execution Server* and *Order Match Engine* to handle different protocols of different markets.

*Authentication Server* handles secure login from client side. *BackOffice Server* communicates with *BackOffice Admin*. *Risk Server* provides functionality for *Risk Manager. Operations Server* interacts with *Operations Support*.

**Solution Blueprint – Feed and Order Handlers**

To connect to different Market Data Engine from different markets, a set of *Feed Handlers* are created as in Figure 21.

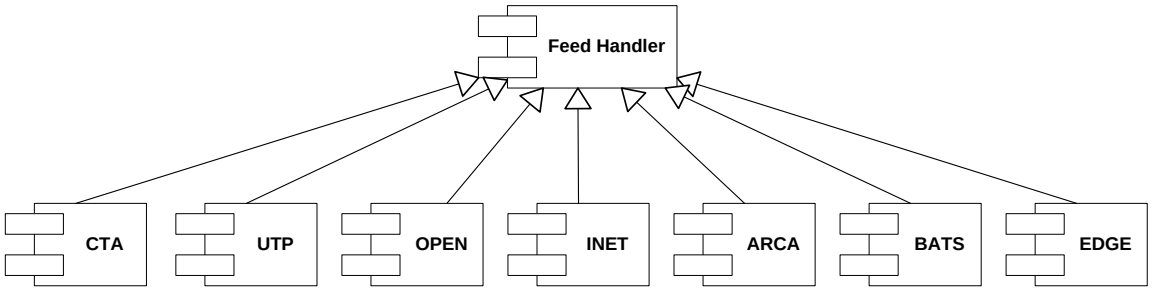


Figure 21: Different Feed Handlers for Different Markets

CTA receives consolidated market data for NYSE and AMEX issues while UTP for NASDAQ issues. OPEN is NYSE exchange, ARCA is the market acquired by NYSE. INET is NASDAQ. BATS and EDGE are other two popular markets.

Figure 22 shows *Order Handlers* for four major markets: NASAQ INET, NYSE ARCA, BATS and EDGE.

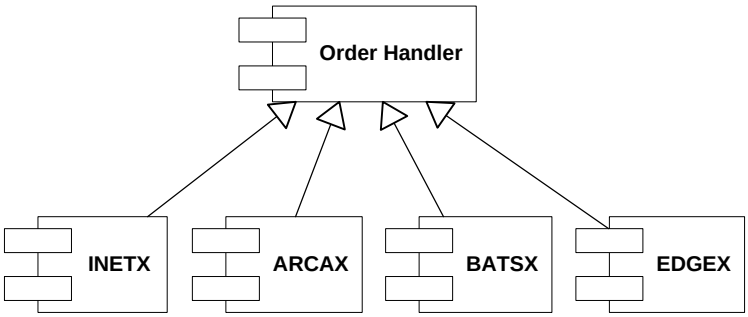


Figure 22: Different Order Handlers for Different Markets

**Solution Blueprint – Mechanisms to Improve Availability**

To improve system availability, every server side components can be deployed as primary or secondary. Primary serves other components most of time and secondary takes over primary in case of primary failure. The secondary can be deployed as hot backup or code backup. This section describes the details of techniques used to for hot backups.

Operations Server knows all the servers and their roles and can switch the roles of any server between primary and secondary. When *Operations Server* detects one primary server does not respond through periodical heartbeat message, it notifies *Operations Support*. Based on the configuration, *Operations Server* can automatically change the standby secondary to take the primary role. Or *Operations Server* tells *Operations Support* the situation and waits for commands to switch the role of specific component.

***Failover of Feed Handlers***

The failover of feed handlers is the simplest of all as market data flows in one direction and no direct connection maintained between provider components and consumer components. Figure 23 illustrates how Feed Handlers fail over.

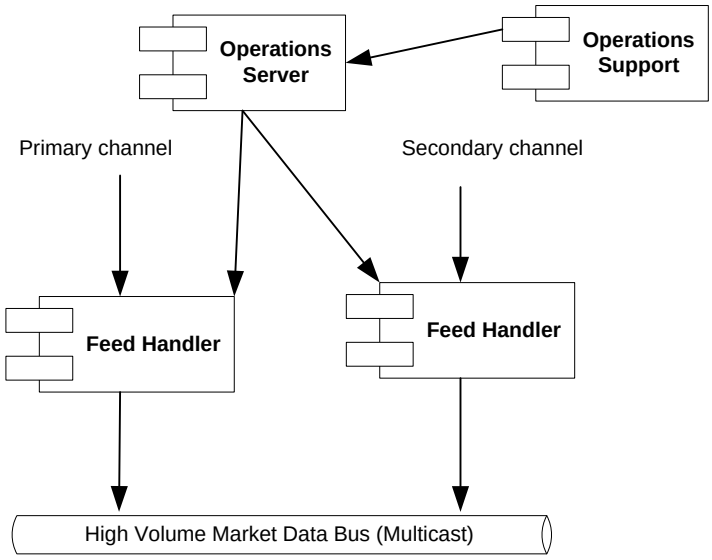


Figure 23: Feed Handler Failover

The same *Feed Handlers* are deployed on two computers. The primary *Feed Handler* receives data from primary channel of *Market Data Engine* while the secondary *Feed Handler* receives from secondary channel. Both primary and second *Feed Handler* process data from incoming channel and connect to the same *High Volume Market Data Bus*. However, only primary *Feed Handler* outputs to *High Volume Market Data Bus*. Both handlers send heartbeats out. On detection of missing heartbeat of primary *Feed Handler, Operations Server* notifies *Operations Support* the urgent event. ***Operations Support***, after verification, decides to put secondary *Feed Handler* to primary and investigate the previous primary *Feed Handler*. Now the new Feed Handler outputs to the

same bus and other components consuming *High Volume Market Data Bus* barely notice the change.

***Failover of Non-handler Server Components***

Most of non-handler server components are a bit harder to fail over than feed handlers as the communications are bidirectional. *Execution Server* is one of these components. Figure 24 illustrates the failover mechanism of *Execution Server*.

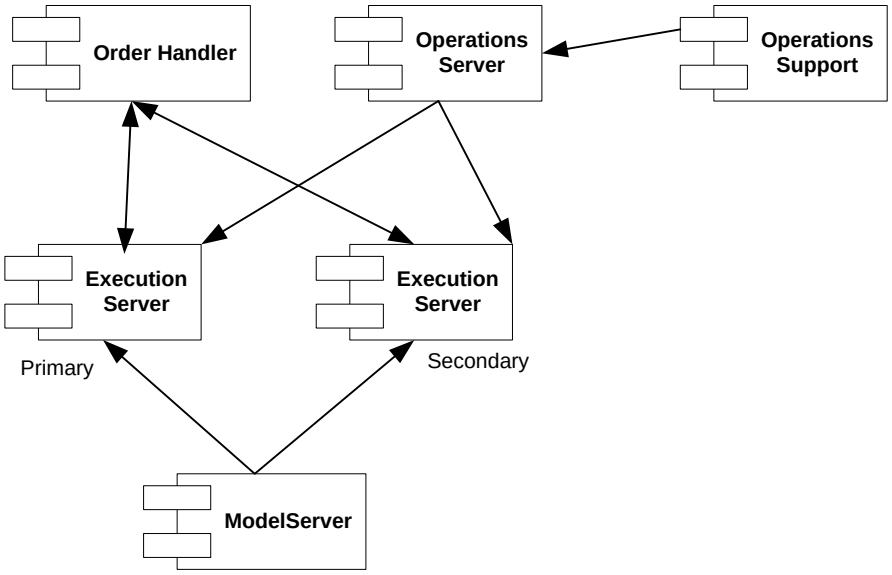


Figure 24: Failover of Non-handler Server Components

*Model Server* connects to both primary and secondary *Execution Servers*. Both of the execution servers connect to the same *Order Handler*. To place an order, *Model Server* sends the order to both execution servers and both of them process it locally. However, only primary *Execution Server* sends the order out to *Order Handler*. The notifications from *Order Handler* are received by both execution servers so that both of

them can update local state of orders. In the event of primary failure, connections between *Order Handler* and *Execution Server* as well as connections between *Model Server* and *Execution Server* are down. However *Order Handler* and *Model Server* do not care as long as there is at least one connection to either *Execution Server* and they send to connections available as if nothing happens. *Operations Support* commands *Operation Server* to switch the role of secondary Execution Server to primary role. Since secondary *Execution Server* keeps states updated as much as primary *Execution Server*, It does not need to synchronize its internal states before serving *Model Server*. The switch takes no time.

***Failover of Order Handlers***

*Order Handlers* failover works a bit more complicated than all other server components because of special characteristics of *Order Match Engine*. Usually connected through a common protocol called FIX (Financial Information Exchange), *Order Match Engine* builds the capability to synchronize *Order Handlers* upon connection. *Order Match Engine* asks *Order Handler* what it knows about its orders and updates *Order Handler* what it does not already have. For this very reason, only one *Order Handler* of the same ID can connect to it. And both primary and secondary *Order Handlers* shares the same ID. For this reason secondary *Order Handler* is not connected to *Order Match Engine*.

As Figure 25 illustrates, to take over the primary role, secondary *Order Handler* makes the connection, synchronized with *Order Match Engine* and then is ready to serve *Execution Servers*.

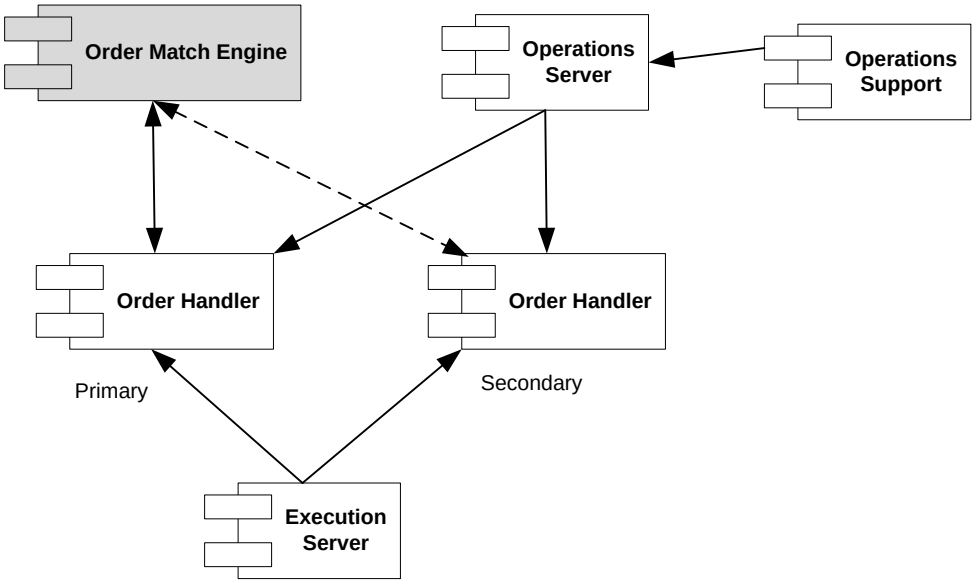


Figure 25: Order Handler Failover

**Solution Blueprint – Mechanisms to Improve Scalability**

Not only can primary server component and secondary server component of the same type co-exist for backup purposes, primary server components themselves can also co-exist to form a server farm each of which serve different groups of clients.

***Scalability Mechanism for Most Server Components***

Figure 26 on the next page illustrates the mechanism for a farm of the same type of server components to server different clients. Upon client login, Authentication Server allocates one instance of each type of server components the client needs to connect according to certain criteria. Because Operations Server has all the information, it communicates with Operations Server to get the best server list for specific client.

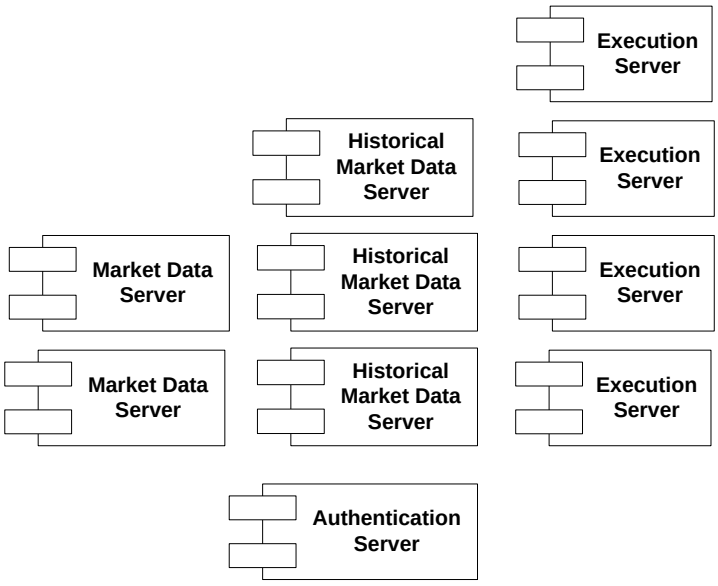


Figure 26: Server Component Farm

***Scalability Mechanism for Authentication Server***

Scalability of *Authentication Servers* requires a special approach as no other server components allocate appropriate *Authentication Servers* to clients. *Load Balancer*, a special network device can serve as the allocator.

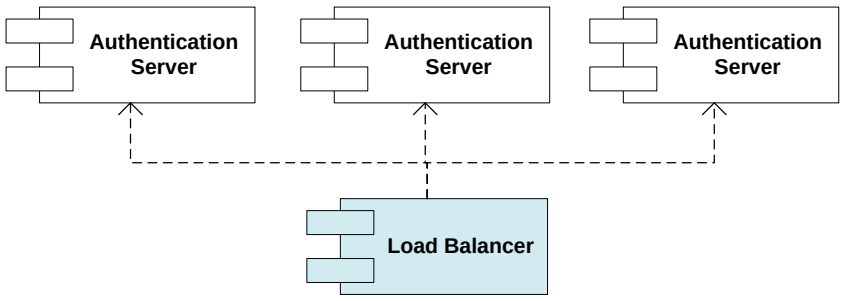


Figure 27: Scalability Mechanism of Authentication Servers

As Figure 27 illustrates, client components connect to the load balancer. Instead of answering the connection request, load balancer redirects the request to one of *Authentication Servers* in the farm which responds to the connection request and allocation is achieved.

**DEPLOYMENT BLUEPRINT**

Deployment Blueprints is all about the allocation of each component to different computer nodes on the network. The client components can be deployed on the same computer node while server components usually are deployed on different computer nodes. The deployment mainly depends on the following factors

- Whether order book support is necessary and what book support is required.

  Most models do not depend on book data but active manual traders usually use book data for precision. Some models analyze order book information for actions.

- Whether connections to multiple execution venue is necessary

  Connection to one execution venue is usually necessary as all venues are inter-connected. However, the advantage to connect to multiple execution venues is performance. *Execution Strategy* can take advantage of multiple execution venues to execute a big order more quickly.

- Whether backups is necessary and what server components require failover

  The critical components, including *Authentication Server*, *Feed Handler, Market Data Server, Historical Market Data Server, Order Handler, Execution Server, Database Writer* and DBMS require failover quickly to minimize interruption to the whole system. Other components can restart manually on the same computer or on other computers to recover. To ensure the smoothest failover in case of component or node failure, every component requires a hot backup to stand by.

- Whether deterioration of service can be tolerated after failover.

  If the system only needs to tolerate one failure at a time, most of the backup components can share one node. However, to keep the backup hot, the computation of the backup component is almost as heavy as the primary component. So the performance is reduced after the switch. The ideal situation is to have all backups on their own computers to avoid the performance hit.

- Whether server component farm is necessary for scalability.

  When clients are more than tens of thousands such as retail brokerage, scalability is important. For a small brokerage with only hundreds of traders and models, one primary of each server component is good enough.

**Minimal Deployment of Server Side Components**

This minimal deployment only has no order book support, only one execution venue, hot backups only for critical components and on the same computer node. Figure 28 on the next page illustrates the minimal scale deployment of the system as an example.

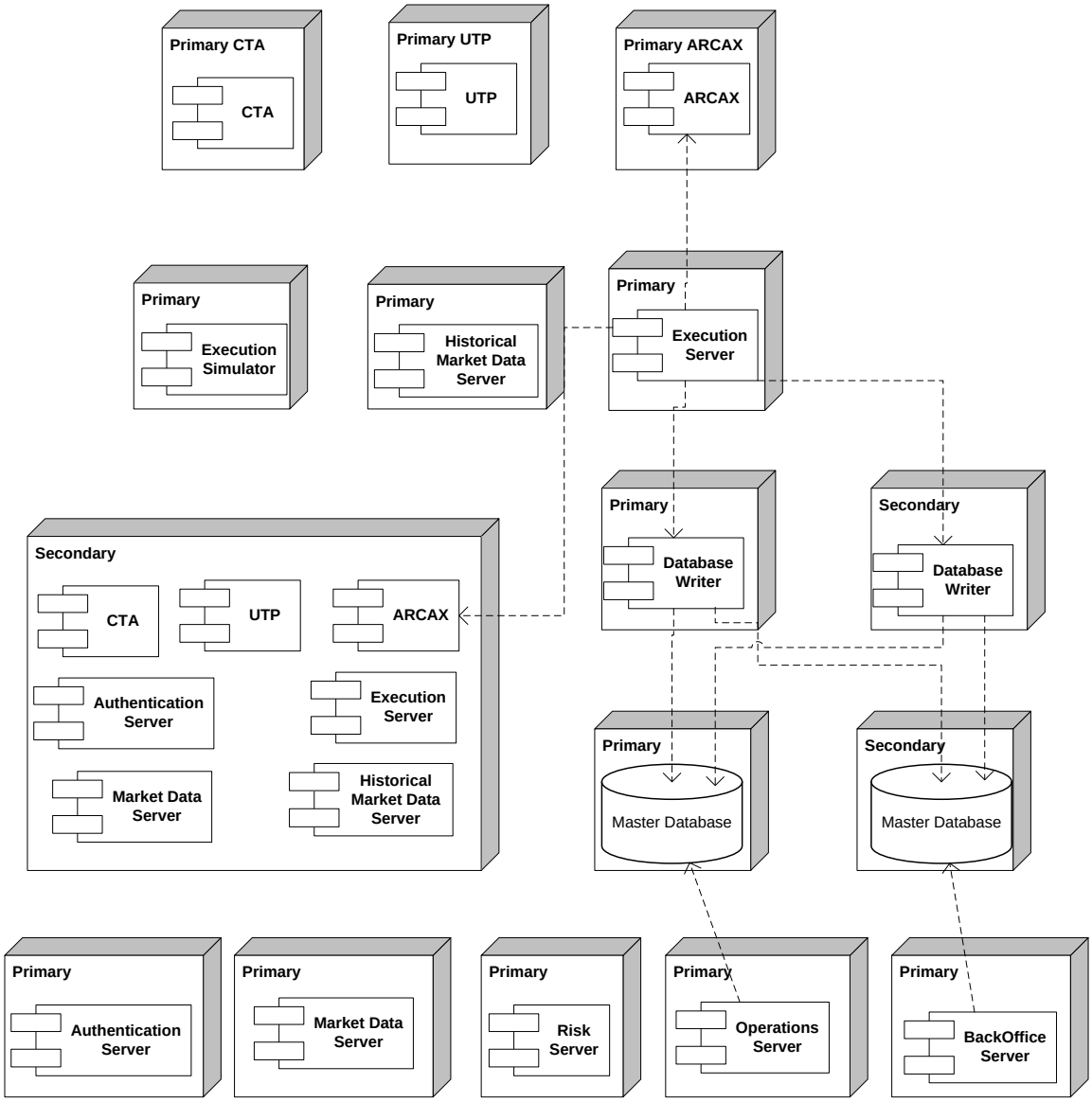


Figure 28: Minimal Deployment of Server Components

# Chapter 4: Research Environment of QUARTS and Model Examples

Research environment provides a friendly environment for quants to design, debug and test quantitative models. As mentioned in the introduction, the quantitative models are written in Python language. The research environment provides a framework to develop models and test it in Python. Many Python environments can be used on both LINUX and Windows. A really good combination for research environment is Eclipse as Integrated Development Environment (IDE), Pypy as JIT interpreter and LINUX as platform according to my test.

The design and implementation of the test framework in Python to support quantitative research is based on Chapter 2. Please review that chapter for terms and concepts. All design diagrams are in UML [15].

### HIGH LEVEL STRUCTURE

Figure 29 on the next page illustrates the high level structure of the framework of research environment. The line without arrow shows a bi-directional association while line with arrow shows one-directional association. In one-directional association, only the class on the root of the arrow can access the class on the point of arrow.

*Account* plays a central role. *MarketDataEngineSimulator* reads recorded market data file (called tick file) and feeds market data to *Account* and *Account* runs the *Model* associated with it. *Account* does all the bookkeeping with the help of *CommissionFeeModel* which provides different commissions and fees charged for more accurate test.

When *Model* detects a signal to take action, it buys or sells a stock through *Account* and *Account* redirects the request to *OrderMatchEngineSimulator*. *Account* maintains all the orders and positions generated by the *Model*. Model has access to *Account* for all the information so that it can make a decision not only from market data, but also from its current positions and working orders. *OrderMatchEngineSimulator* simulates execution of orders based on the market data fed by *MarketDataEngineSimulator*.

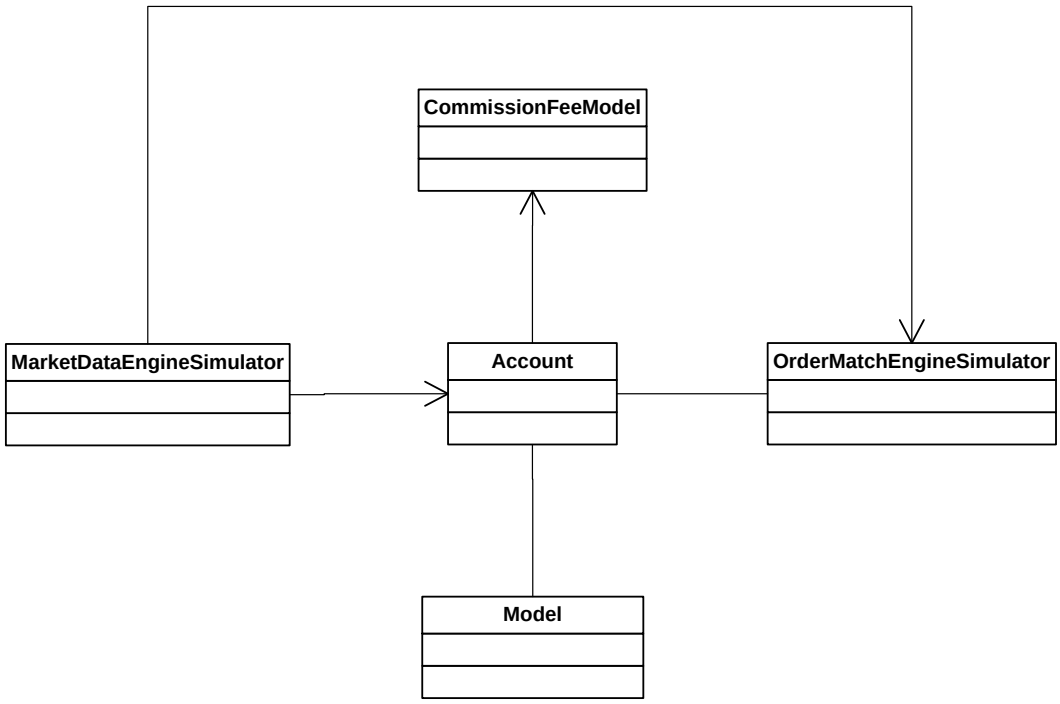


Figure 29: High Level Structure

## MODEL

Figure 30 illustrates the components of *Model*. First, *Model* works like a template with options and parameters. It groups entry, exit, position sizing and risk management strategies together. This version does not cover execution strategy. Instances of each strategy are created for each symbol and held by a *ModelInstanceOnSymbol* instance. The reason to have one strategy instance per symbol is that the same strategy may maintain different states for different symbols.

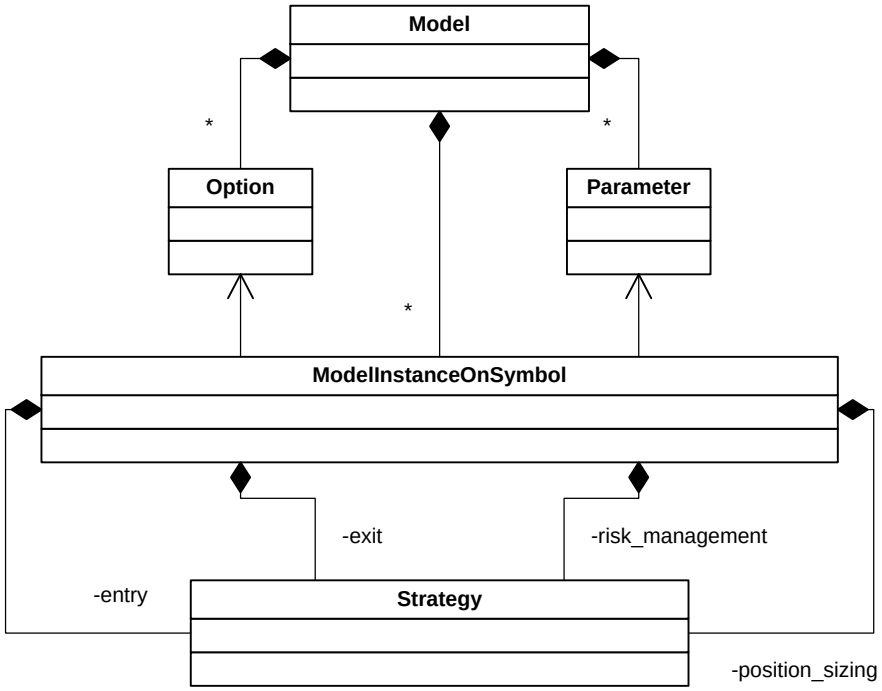


Figure 30: Model Components

## ACCOUNT

*Account* isolates the associated Model from *MarketDataEngineSimulator* and *OrderMatchEngineSimulator*. It feeds the associated model with market data and

captures orders generated from the model. This frees *Model* from order interaction, cash management and position management so that *Model* can focus on order generation only. The current state of cash, orders and positions can be access by Model for advanced strategies.

Figure 31 illustrates the details of *Account*. It contains *CashFlowManager* which logs every cash related transaction including orders, commissions and fees as well as other special cash transactions. *PositionManager* maintains positions aggregated by symbols. It uses Fist-In-First-Out algorithm to match order executions and positions. *AggregatePosition* also maintains realized profit and loss (PnL) for each symbol. Each individual *Position* maintains up guard price and down guard price which are used by exit strategies. *Account* maintains order status through *OrderManager* by three order lists: working orders, filled orders and cancelled orders.

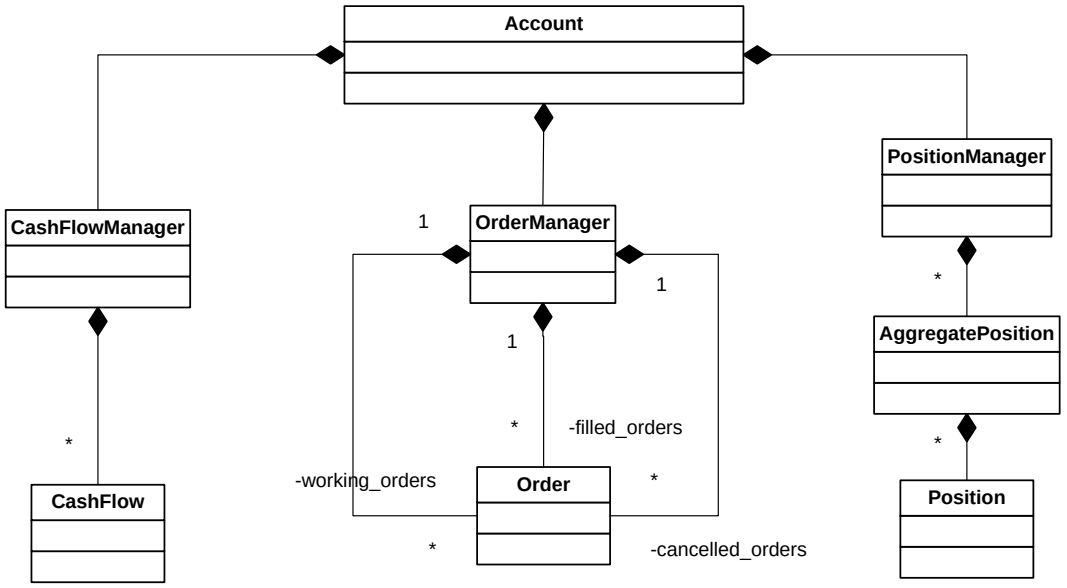


Figure 31: Account Details

### MARKET DATA ENGINE AND ORDER MATCH ENGINE SIMULATOR

*MarketDataEngineSimulator* associates *MarketDataParser* which parses market data file to market data, *OrderMatchEngineSimulator*, which receives market data feed to simulate order execution and *Accounts* which run market data through associated *Models*. *MarketDataEngineSimulator* itself maintains a map from symbol to its current *MarketData* so that certain statistics can be maintained throughout the day. Market data are also fed to Chart component to generate bars of different types and intervals during the day. Figure 32 illustrates their relationship.

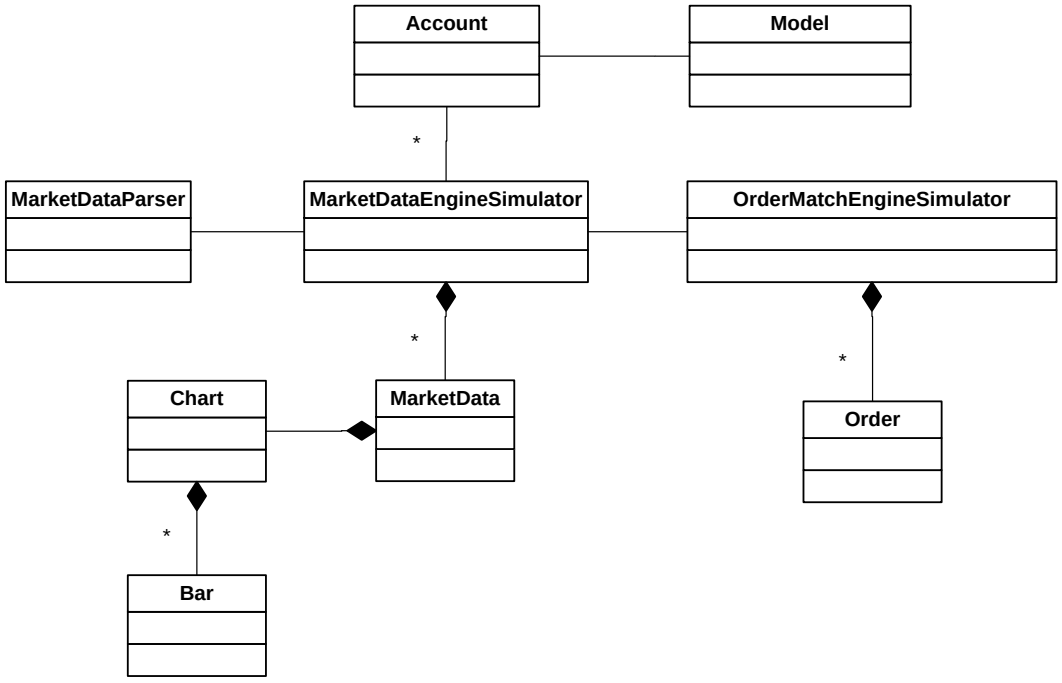


Figure 32: Market Data Engine and Order Match Engine Simulator

## STRATEGY LIBRARY

Models are composed of different components called strategies each of which covers a specific area of the model. Although models cannot be shared, the strategies of the model can. In fact, except entry strategy that characterizes model, the rest of strategies are usually shared. To illustrate the point, I have created a few commonly used strategies in each category: exit strategy, position sizing strategy and risk management strategy.

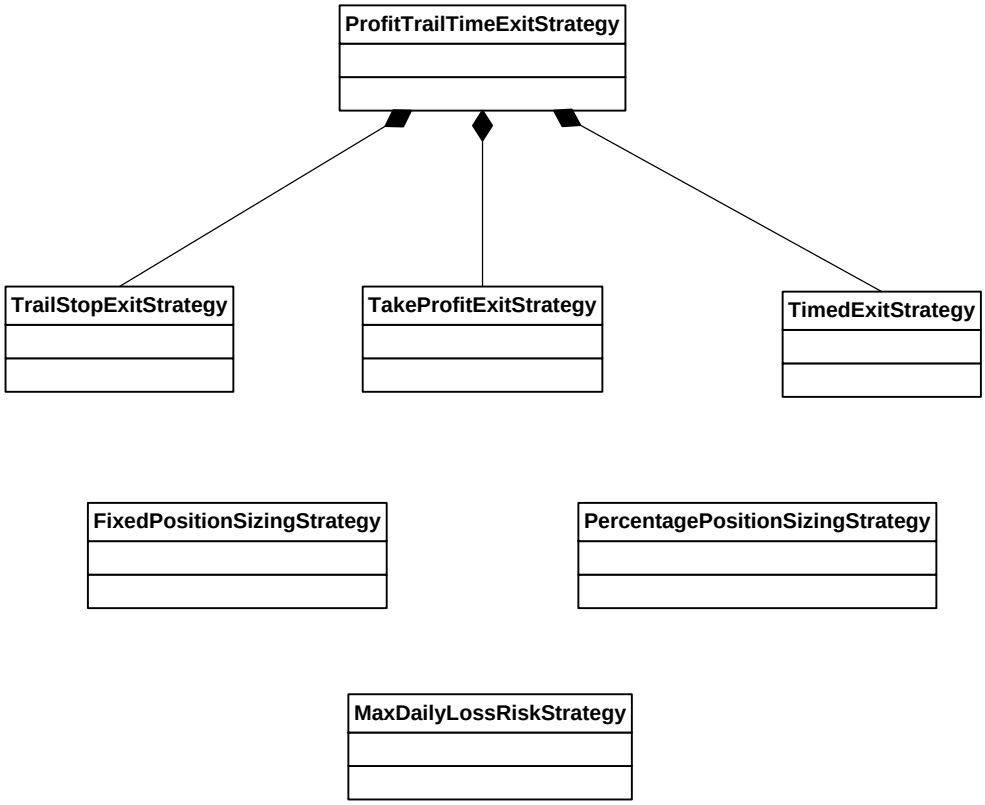


Figure 33: Strategy Library

Figure 33 illustrates a few strategies in strategy library. *TrailStopExitStrategy* is a popular exit strategy used by many manual traders. Given a dollar trail offset or

percentage trail offset, it sets up the protective price point (stop price). For a long position, it is an offset below the position entry price. It triggers an order to sell the position when market price breaches down the stop price. Stop price stays the same when market price moves down but moves up along with market price. It works the same for a short position but the stop price is above the entry point and the breach is to the upside.

*TakeProfitExitStrategy* triggers orders to close positions when the market price reaches certain profit target, in dollar amount or percentage of the entry prices. *TimedExitStrategy* is a simple exit strategy which triggers orders to close positions at specific time. It is useful for most intraday models that close all positions for the day.

*FixedPositionSizingStrategy* is a simple strategy that returns the fixed shares user specifies in parameters. It checks account to make sure the account has enough capital for the trade. *PercentagePositionSizingStrategy*, on the other hand, specifies the fixed dollar amount as the percentage of initial capital of the account so that each trade has different shares but a similar dollar amount.

*MaxDailyLossRiskStrategy* is the simplest of all risk management strategies. It basically monitors the total loss both realized and unrealized for the day. When it reaches the preset amount the model stops opening positions and all the existing ones are liquidated immediately.

### MARKET DATA FILE

Market data file, an input to Python research environment, is recorded from live market data. Live market data are big (more than 10G a day and growing) and come in high rates in the form of unreliable UDP multicast. Sometimes data source, network or application issues lead to gaps in recorded market data files. However, a good back-test system requires high quality market data. In reality, the market data files are bought from specialized data vendors to ensure quality. The market data files, or tick files used in the

system are such high quality files. The file contains trade messages in text format where each row represents one trade message with NBBO information. Each trade message contains pipe-delimited fields: date, time, symbol, price, shares, trade conditions, best bid, best ask, bid size, ask size, among others.

Market data files of five trading days between 4/15/2013 and 4/19/2013 are used to evaluate quantitative model examples and to measure system performance. Characteristics of these files are shown in Table 11.

| Date | File Size (bytes) | Symbols | Trade Messages |
|---|---|---|---|
| 4/15/2013 | 2,857,730,829 | 7,420 | 31,216,807 |
| 4/16/2013 | 2,267,676,959 | 7,392 | 24,789,312 |
| 4/17/2013 | 2,762,625,656 | 7,376 | 30,237,655 |
| 4/18/2013 | 2,513,296,838 | 7,311 | 27,557,536 |
| 4/19/2013 | 2,184,532,712 | 7,319 | 23,927,411 |
| Average | 2,517,172,599 | 7,364 | 27,545,744 |

Table 11: Characteristics of Market Data Files

### RANDOM WORK MODEL EXAMPLE

There is a theory about randomness on Wall Street [5]. I would like to create a model to test how a random trade turns out. My test is simple. With a $1 million account, on 10:30 am ET of a trading day, for one minute I look across the market for 100 stocks between $10 and $100 with liquidity tested by volume over 1 million shares during the first trading hour. I toss a coin and get into the trade with $10,000 (rounded to multiples of 100 shares) each on heads up. Then I set up a profit target of 3% and a trail stop of 1%

with risk reward ratio 3 to 1. I stop the model and liquidate the account when loss accumulates to more than $3000. If the positions remain open on 15:55, 5 minutes before the close, I close all of them. I run the strategy for both long side and short side. Figure 34 shows the program in Python that runs Random Walk model.

```
def main(tick_file, symbol_list):
    c = PerShareCommissionModel('1cPerShare',-0.01)
    x = OrderMatchEngineSimulator()
    d = MarketDataEngineSimulator()
    p={ 'PercentShares':1.0, 'ProfitPercent':3, 'TrailPercent':1,\
        'ExitTime':time(15,55), 'MaxDailyLoss':3000.0}
    lo={'Long':True, ,'MinVolume':1000000,\
        'StartTime':time(10,30),'StopTime':time(10,31),'LowPrice':10.0,'HighPrice':100.0}
    rwlm = RandomWalkModel('RandomWalkLongModel', symbol_list, lo, p)
    rwla = Account('RandomWalkLongAccount', 1000000, long=True)
    rwla.associate(rwlm,x,c)
    so={'Long':False, ,'MinVolume':1000000,\
        'StartTime':time(10,30),'StopTime':time(10,31),'LowPrice':10.0,'HighPrice':100.0}
    rwsm = RandomWalkModel ('RandomWalkShortModel', symbol_list, so, p)
    rwsa = Account('RandomWalkShortAccount', -1000000, long=False)
    rwsa.associate(rwsm,x,c)
    d.run(x,[rwla, rwsa])
```

Figure 34: Program to Run Random Walk Model

Random walk model has a random walk entry strategy and standard exit, risk and position sizing strategies from strategy library. Random walk entry strategy supports both long and short. Figure 35 on next page shows random walk entry strategy and random walk model.

```
class RandomWalkEntryStrategy(Strategy):
    def __init__(self, data, account, options, params):
        Strategy.__init__(self, data, account, options, params)
        self.tossed = False
    def evaluate(self, data, account, model_instance):
        if account.state != Account.ACTIVE_STATE:
            return False
        if data.time < self.start_time or data.time >= self.stop_time\
            or data.price < self.low_price or data.price >= self.high_price\
            or data.acc_volume < self.min_volume:
            return False

        if self.tossed:
            return False
        self.tossed = True

        odd = random()
        if odd < 0.5:
            return False

        shares = model_instance.position_sizing.evaluate(data, account, model_instance)
        if shares == 0:
            return False

        if self.long:
            account.buy(data.symbol, shares, data.price)
        else:
            account.short(data.symbol, shares, data.price)

class RandomWalkModel(Model):
    def __init__(self, name, symbolList, options, params):
        Model.__init__(self, name, symbolList, options, params)
    def create_entry_strategy(self, data):
        return RandomWalkEntryStrategy(data, self.account, self.options, self.params)
    def create_exit_strategy(self, data):
        return ProfitTrailTimeExitStrategy(data, self.account, self.options, self.params)
    def create_risk_management_strategy(self,data):
        return MaxDailyLossRiskStrategy(data, self.account, self.options, self.params)
    def create_position_sizing_strategy(self,data):
        return PercentagePositionSizingStrategy(data, self.account, self.options, self.params)
```

Figure 35: Random Walk Model & Its Entry Strategy

The model has run for a week with five trading days between 4/15/2013 and 4/19/2013. The characteristics of market data files on these days are shown in Table 11. Model results are shown in Table 12 where

*PnL*: profit and loss

*Entry*: signals triggered by entry strategy

*Fee* : commissions and fees to perform the trade

*Time*: time spent to run mode with market data file.

| Date | Random Walk Model Long | | | | Random Walk Model Short | | | |
|---|---|---|---|---|---|---|---|---|
| | PnL | Entry | Fee | Time | PnL | Entry | Fee | Time |
| 4/15 | -3,360.77 | 226 | 666 | 5:41 | 12,573.53 | 233 | 698 | 6:42 |
| 4/16 | 1,112.45 | 226 | 700 | 4:58 | -3,273.51 | 230 | 714 | 4:43 |
| 4/17 | -3,256.32 | 228 | 750 | 5:34 | 2,916.01 | 230 | 712 | 6:14 |
| 4/18 | -3,416.40 | 221 | 732 | 4:59 | -3,326.32 | 216 | 722 | 5:07 |
| 4/19 | 3,384.96 | 228 | 734 | 4:47 | -3,385.18 | 224 | 726 | 4:14 |
| Sum | -5,536.08 | 1,129 | 3,582 | | 5,531.53 | 1,123 | 3,572 | |

Table 12: Random Walk Model Long and Short Results

Among ten model runs, six of them end up with liquidation by risk management strategy with loss more than $3000 (actual loss is higher because of commissions and slippage). Four of them end up with gains with 4/15/2013 short for maximum gain of $12,573.53. By the end of the week, we lost $4.55 (-5,536.08+5,531.53) after commissions. The commission cost of $7,154 (3,582+3,572) ate all the profits.

Without surprise, this pure random model does perform randomly even with limited testing of a few days worth of data. It is one thing to recognize that stock market is random in some sense. It is another to make money out of it. It is the quants' job to find what is relatively certain with random walk. Before back-testing it with years' worth of data, the refinement of the model is necessary.

## MOMENTUM MODEL EXAMPLE

This is a very simple momentum model. I look at a 5-minute chart of any symbol in my range between $10 and $100 with enough liquidity tested by volume no less than 100,000 shares. If there are three consecutive up bars on the chart, I go long it for 100 shares. Three consecutive down bars in a row trigger a short of 100 shares. The exit and risk management strategies are the same as random walk model. The strategy starts at 9:45am and stops at 15:45pm. The model closes all open positions at 15:55pm. Figure 36 shows the main program. Figure 37 on the next page shows momentum entry strategy and the model. Momentum model supports both long and short.

```
def main(tick_file, symbol_list):
    c = PerShareCommissionModel('1cPerShare',-0.01)
    x = OrderMatchEngineSimulator()
    d = MarketDataEngineSimulator('5minute')
    p={'ChartType':'5minute' 'MomentumBars':3, 'FixedShares':100
        'ProfitPercent':3,'TrailPercent':1,'ExitTime':time(15,55),MaxDailyLoss':3000.0}
    lo={'Long': True, ,'MinVolume':100000\
        'StartTime':time(9,45),'StopTime':time(15,45),'LowPrice':10.0,'HighPrice':100.0}
    mlm = MomentumModel('MomentumLongModel', symbol_list, lo, p)
    mla = Account('MomentumLongAccount', 1000000, long=True)
    mla.associate(mlm,x,c)
    so={'Long':False, ,'MinVolume':100000\
         'StartTime':time(9,45),'StopTime':time(15,45),'LowPrice':10.0,'HighPrice':100.0}
    msm = MomentumModel ('MomentumShortModel', symbol_list, so, p)
    msa = Account('MomentumShortAccount', -1000000, long=False)
    msa.associate(msm,x,c)
    d.run(x,[mla,msa])
```

Figure 36: Program to run Momentum Model

.

```
class MomentumEntryStrategy(Strategy):
    def __init__(self, data, account, options, params):
        Strategy.__init__(self, data, account, options, params)
        self.momentum_bars = params['MomentumBars'] if 'MomentumBars' in params else 3
        self.chart_type = params['ChartType'] if 'ChartType' in params else '5minute'
    def evaluate(self, data, account, model_instance):
        if account.state != Account.ACTIVE_STATE\
            or account.order_manager.has_working_order(data.symbol)\
            or account.position_manager.has_position(data.symbol):
            return False
        if data.time < self.start_time or data.time >= self.stop_time\
            or data.price < self.low_price or data.price >= self.high_price\
            or data.acc_volume < self.min_volume:
            return False
        try:
            chart = data.charts[self.chart_type]
        except KeyError:
            return False
        if len(chart.bars) <= self.momentum_bars: # not including forming bar
            return False
        direction = chart.bars[-2].direction()
        if direction == 0 or (direction > 0 and not self.long) or (direction < 0 and self.long):
            return False
        for i in xrange(-3, -self.momentum_bars-2, -1):
            if chart.bars[i].direction() != direction:
                return False
        shares = model_instance.position_sizing.evaluate(data, account, model_instance)
        if shares == 0:
            return False;
        if self.long:
            account.buy(data.symbol, shares, data.price) # market order, day TIF
            return True
        else:
            account.short(data.symbol, shares, data.price)
            return True

class MomentumModel(Model):
    def __init__(self, name, symbolList, options, params):
        Model.__init__(self, name, symbolList, options, params)
    def create_entry_strategy(self, data):
        return MomentumEntryStrategy(data, self.account, self.options, self.params)
    def create_exit_strategy(self, data):
        return ProfitTrailTimeExitStrategy(data, self.account, self.options, self.params)
    def create_risk_management_strategy(self,data):
        return MaxDailyLossRiskStrategy(data, self.account, self.options, self.params)
    def create_position_sizing_strategy(self,data):
        return FixedPositionSizingStrategy(data, self.account, self.options, self.params)
```

Figure 37: Momentum Model & Its Entry Strategy

Table 13 shows the results of momentum model running both long and short for a week with five trading days between 4/15/2013 and 4/19/2013. Characteristics of market data files on these days are shown in Table 11. Among 10 runs, 7 end up with loss more than $3000 and were stopped by risk management strategy. 3 end up with profits. And one of them is big: $15,974.66. At the end of week it lost $506.41 (-2,430.98+1,924.57). The commissions cost $14,501 (4,835+9,666) which is significant.

| Date | Momentum Model Long | | | | Momentum Model Short | | | |
|---|---|---|---|---|---|---|---|---|
| | PnL | Entry | Fee | Time | PnL | Entry | Fee | Time |
| 4/15 | -3,511.65 | 465 | 929 | 6:18 | -3,220.35 | 348 | 697 | 5:50 |
| 4/16 | 3,767.73 | 673 | 1,346 | 6:45 | -3,608.49 | 667 | 1,335 | 4:56 |
| 4/17 | -3,282.63 | 339 | 678 | 5:14 | -3,896.82 | 633 | 1,266 | 5:52 |
| 4/18 | -3,154.27 | 218 | 436 | 5:09 | -3,324.42 | 287 | 574 | 5:00 |
| 4/19 | 3,739.84 | 723 | 1,446 | 6:40 | 15,974.66 | 2,897 | 5,794 | 17:50 |
| Sum | -2,430.98 | 2,418 | 4,835 | | 1,924.57 | 4,832 | 9,666 | |

Table 13: Momentum Model Long and Short Results

I am surprised that a model as simple as this does not lose much money after a week's run. I would be more surprised if such a strategy can make money consistently day in and day out. The commissions cost very much. Nevertheless, a model that really works in the market starts from many simple ideas such as this one. Refinement of the model should lower the times the entry signal triggers to avoid unnecessary commission costs and at the same time improve the winning rate.

## PERFORMANCE MEASUREMENTS

Although Python research environment is far slower than back-test, forward-test and living trading environment, it is still valuable to see upper bounds of the time required to evaluate a model written in Python language.

The environment is Dell Precision T5400 with Ubuntu 12.04. It has four Xeon X5450 CPU @ 3.00 GHz with 3.25GB of RAM. Since the python program is single-threaded, only one CPU is actually used. It runs for the week from 4/15/2013 to 4/19/2013 with five trading days characterized in Table 11. Each day has average 25 million messages with 7,364 symbols. Each model is deployed to all symbols in the market.

The measurements are taken on two models illustrated in previous sections with small variations of parameters. Table 14 gives the details of each model involved for measurements.

| Model | Name | Description |
|---|---|---|
| 1 | RandomWalkLong100 | Long 100 shares on each entry signal |
| 2 | RandomWalkShort100 | Short 100 shares on each entry signal |
| 3 | RandomWalkLong1Percent | Long 1% of initial capital on each entry signal |
| 4 | RandomWalkShort1Percent | Short 1% of initial capital on each entry signal |
| 5 | MomentumLong100 | Long 100 shares on each entry signal |
| 6 | MomentumShort100 | Short 100 shares on each entry signal |
| 7 | MomentumLong1Percent | Long 1% of initial capital on each entry signal |
| 8 | MomentumShort1Percent | Short 1% of initial capital on each entry signal |

Table 14: Models for Performance Measurements

I have set up five passes for each day to evaluate a group of these models together on each pass as shown in Table 15. From each pass I derive the average time to evaluate one model.

| Pass | Model Count | Models Involved |
|---|---|---|
| 0 | 0 | 0, Test the file reading and parsing performance for baseline |
| 1 | 1 | Model 1 |
| 2 | 2 | Model 1-2 |
| 3 | 4 | Model 1-4 |
| 4 | 8 | Model 1-8 |

Table 15: Measurement Passes

Table 16 shows the measurement results for each pass with Pypy 2.0. Average time spent on each Random Walk model and Momentum model is derived as following:

*RW1 = P1 – P0; RW2 = (P2 – P1) / 2; RW4=(P3-P0)/4; M4 = (P4-P3)/4*

| Pass | Models | 4/15/2013 | 4/16/2013 | 4/17/2013 | 4/18/2013 | 4/19/2013 | Average |
|---|---|---|---|---|---|---|---|
| P0 | 0 | 00:04:50 | 00:03:45 | 00:04:42 | 00:04:16 | 00:03:44 | 00:04:15 |
| P1 | 1 | 00:06:01 | 00:04:37 | 00:05:42 | 00:05:09 | 00:04:27 | 00:05:11 |
| P2 | 2 | 00:06:33 | 00:05:11 | 00:06:17 | 00:05:55 | 00:05:11 | 00:05:49 |
| P3 | 4 | 00:07:32 | 00:06:07 | 00:07:28 | 00:06:24 | 00:05:52 | 00:06:40 |
| P4 | 8 | 00:13:42 | 00:12:48 | 00:10:39 | 00:09:13 | 00:25:38 | 00:14:24 |
| RW1 | | 00:01:11 | 00:00:52 | 00:01:00 | 00:00:53 | 00:00:43 | 00:00:55 |
| RW2 | | 00:00:51 | 00:00:43 | 00:00:47 | 00:00:49 | 00:00:43 | 00:00:47 |
| RW4 | | 00:00:40 | 00:00:35 | 00:00:41 | 00:00:32 | 00:00:32 | 00:00:36 |
| M4 | | 00:01:32 | 00:01:40 | 00:00:47 | 00:00:42 | 00:04:56 | 00:01:55 |

Table 16: Performance Measurements with Pypy

The same measurements are done with Python 2.7 too in Table 17.

| Pass | Models | 4/15/2013 | 4/16/2013 | 4/17/2013 | 4/18/2013 | 4/19/2013 | Average |
|---|---|---|---|---|---|---|---|
| P0 | 0 | 00:13:36 | 00:10:54 | 00:13:10 | 00:11:59 | 00:10:21 | 00:21:00 |
| P1 | 1 | 00:20:16 | 00:15:59 | 00:19:35 | 00:17:59 | 00:15:39 | 00:17:53 |
| P2 | 2 | 00:25:18 | 00:20:29 | 00:24:21 | 00:22:22 | 00:19:34 | 00:22:24 |
| P3 | 4 | 00:35:06 | 00:28:23 | 00:34:00 | 00:31:13 | 00:27:26 | 00:31:13 |
| P4 | 8 | 01:07:44 | 01:06:23 | 00:54:16 | 00:49:01 | 02:02:29 | 01:11:58 |
| RW1 | | 00:06:40 | 00:05:05 | 00:06:25 | 00:06:00 | 00:05:18 | 00:05:53 |
| RW2 | | 00:05:51 | 00:04:47 | 00:05:35 | 00:05:11 | 00:04:36 | 00:05:12 |
| RW4 | | 00:05:22 | 00:04:22 | 00:05:12 | 00:04:48 | 00:04:16 | 00:04:48 |
| M4 | | 00:08:09 | 00:09:30 | 00:05:04 | 00:04:27 | 00:23:45 | 00:10:11 |

Table 17: Performance Measurements with Python 2.7

From the measurements we can conclude:

- Python is many times slower than Pypy. Pypy takes 14:24 to run eight models for a day while Python takes 01:11:58 to do the same. Pypy is around 5 times faster. Also, Pypy takes an average of 01:55 to evaluate a typical model as Momentum model while Python takes 10:11, around 5 times slower. Just-In-Time (JIT) compiler in Pypy works perfectly for scenarios of model evaluation where the same code is called around 15 million times a day. Pypy is the clear choice to be embedded in production.
- It takes more time to parse the file than to evaluate the model. In Pypy, the average parsing time is (P0) 00:04:15 while evaluations of Momentum model takes 00:01:55. QUARTS back-test environment parses market data file in C++ a lot faster so we should not worry about the Python parsing time.

- It takes significantly more time to run Momentum model (00:01:55) than Random Walk model (00:00:36). Momentum model evaluates 5-minute charts for major entry signal while Random Walk model depends on a flip of a coin. It shows that more complicated model will take a lot more time to evaluate. Real models are quite more complicated than Momentum models. We should pay attention to evaluate each individual model so that it should not take too much time in production.
- Average time to evaluate one Momentum model for full day's data (06:30:00, from 9:30 to 16:00) is 00:01:55. Given 33% load, we should be able to evaluate 65 (06:30:00 / 3 / 00:01:55) models similar to Momentum model deployed to all 7000 symbols. If the actual model is only limited to a list of symbols of 1000, we should be able to run 455 (65*7) such models. If the real model is 5 times more complicated than Momentum model, we can still run 90 such models.

One outlier in Table 16 gives us some concern. It takes 00:04:56 to run the same Momentum model on 4/19/2013 while it takes only 00:01:40 to do the same on 4/17/2013. Detailed investigation shows that Model 6, MomentumShort100 runs 00:12:02 for the day. Continued investigation reveals it triggered 2,897 entry signals on 4/19/2013 and 667 on 4/16/2013. The process of entry signals takes some time in Python. To prove this is the issue, I ran the same model on 4/19/2013 with only NASDAQ 100 components and it only took 00:00:34 to evaluate with 115 entries. I also tried to run the same model on the same day and limited it to 1000 entries. And it takes 00:04:17 to do so. It proves the entries handling time is as significant as evaluation for the model.

In reality, a model should have an upper limit of entry signals for one day. A model triggering more than 1000 entry signals is clearly too high frequent to be built in Python Language. A C++ equivalent should be implemented. A normal model deployed

to a symbol list less than 500 and triggers two entries on average for each symbol does not break 1000 entry signals upper limit.

Also the processing of entry signals measured here are in Python and C++ implementation should be a lot faster ([16] shows C++ is 8 times faster than Python with JIT). The production environment should perform a lot better than measurements described in this section.

# Summary

In this report, we analyzed the core of quantitative model and designed the architecture of a quantitative research and trading system (QUARTS). We implemented the research environment of the system in Python with two sample models.

**FUTURE WORK**

- We would like to integrate fundamentals information dataset, the information in corporate earnings report, into model environment.
- We would like to integrate corporate action dataset into the system for chart adjustment based on splits.
- The current implementation only loads one day's market data and generate minute bars. It would be better for quantitative models to analyze intra-day charts for the past a few days but only to act on the current day market data. After integration of corporate actions dataset we would like to extend intraday chart from one day to multiple days (30 days or 60 days) and adjust prices to account for splits accordingly.
- We would like to pre-process tick files to generate daily charts for all available trading days we have, adjust prices according to corporate action information and load those bars before the test date for quantitative models to access.
- We would like to integrate technical analysis library from TA-LIB [17] for Python into the system so that quantitative models can be based on these technical studies.
- We would like to implement C++ framework for back-test, forward-test and living trading in LINUX environment.

- Execution strategy, the strategy to execute orders smartly as an optional component of quantitative model, is analyzed in Chapter 2 but is not implemented. We would like to integrate it into quantitative model in the future.
- We would like to add charts with bars accumulated by ticks, volume and price ranges. Just as market data during every minute generate 1-minute bars, market data for every 1000 trades generate 1000-tick bars and market data for every 100,000 shares generate 100,000-volume bars. Also market data with prices in a range of $1 in continuous in time generate $1-price range bars.

## Appendix: Software Architecture and Design Methodologies

In the 1992 seminal article *Foundations for the Study of Software Architecture* [6], after examining the architectures of the other disciplines such as hardware, networks, and buildings, Perry and Wolf defined with insight:

Software Architecture = {Elements, Form, Rationale},

or, a set of architectural (or, if you will, design) elements that have a particular form. Elements are processing, data, or connecting elements. Form is defined in terms of the properties of, and the relationships among, the elements – that is, the constraints on the elements. The rationale provides the underlying basis for the architecture in terms of the system constraints, which most often derive from the system requirements.

Garlan and Shaw in their paper *An Introduction to Software Architecture* [7] described software architecture as "…beyond the algorithms and data structures of the computation: designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives." "… to treat an architecture of a specific system as a collection of computational components – or simply components – together with a description of the interactions between these components – the connectors."

Hayes-Roth defined software architecture in *Architecture-Based Acquisition and Development of Software: Guidelines and Recommendations from the ARPA Domain-Specific Software Architecture (DSSA) Program* [8] as: "…An abstract system

specification consisting primarily of functional components described in terms of their behaviors and interfaces and component-component interconnections. The interconnections provide means by which components interact. Architectures are usually associated with a rationale that documents and justifies constraints on component and interconnections or explains assumptions about the technologies which will be available for implementing application consistent with architecture."

C. Gacek, A. Abd-Allah, B. Clark and B. Boehm made an effort to clarify the definition in their paper *On the definition of Software System Architecture* [9]: "A software system architecture comprises:

- A collection of software and system components, connections, and constraints.
- A collection of system stakeholders' requirement statements.
- A rationale which demonstrates that the components, connections, and constraints define a system that, if implemented, would satisfy the collection of system stakeholders' requirement statements.

And finally, in *Software Architecture in Practice* [10], L. Bass, P. Clements, and R. Kazman defined what does constitute software architecture:

> "The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them."

The definition implies

- Architecture defines software elements. The architecture embodies information about how the elements relate to each other. Thus an architecture is foremost an abstraction of a system that suppresses details of elements that do not affect how they use, are used by, related to or interact with other elements. To put it the other way, element details – those having to do solely with internal implementations – are not architectural. By intention, the definition does not specify what the architectural elements and relationships are. Is a software element an object? A process? A library? A database? A commercial product? It can be any of these things and more.
- That systems can and do comprise more than one structure and that no one structure can irrefutably claim to be the architecture. This emphasizes that different architectural views of the same system are together the architecture.
- That every computing system with software has software architecture because every system can be shown to comprise elements and the relations among them. However, it does not necessarily follow that the architecture is known to anyone even though every system has architecture. This reveals the difference between the architecture of a system and the representation of that architecture. Unfortunately, architecture can exist independently of its description or specification, which raises the importance of architecture documentation and architecture reconstruction.
- The behavior of each element is part of the architecture insofar as that behavior can be observed or discerned from the point of view of another element. Such behavior is what allows elements to interact with each other. Such behavior is

what allows elements to interact with each other, which is clearly part of the architecture. It does not mean that the exact behavior and performance of every element must be documented in all circumstances; however, to the extent that an element's behavior influences how another element must be written to interact with it or influences the acceptability of the system as a whole, this behavior is part of the software architecture.

- That it is indifferent as to whether the architecture for a system is a good one or a bad one, meaning that it will allow or prevent the system from meeting its behavioral, performance, and life-cycle requirements. This raises the importance of architecture evaluation and architectural design.

**REPRESENTATION**

Many times the software architecture suffers from the fact that the designer puts more information than practical. On the other hand, as an important communication tool to discuss with system stakeholders, one structure is not good enough to fit all stakeholders' perspectives. The following four methods categorize views of the architecture for the same system.

### 4+1 Views

*Architectural Blueprints - The 4+1 View Model of Architecture* [11] by P. Kruchten in 1995 ventured into the idea to describes the same system in five concurrent views, each of which addresses a specific set of concerns of interest to different stakeholders in the system as is illustrated in Figure 38:

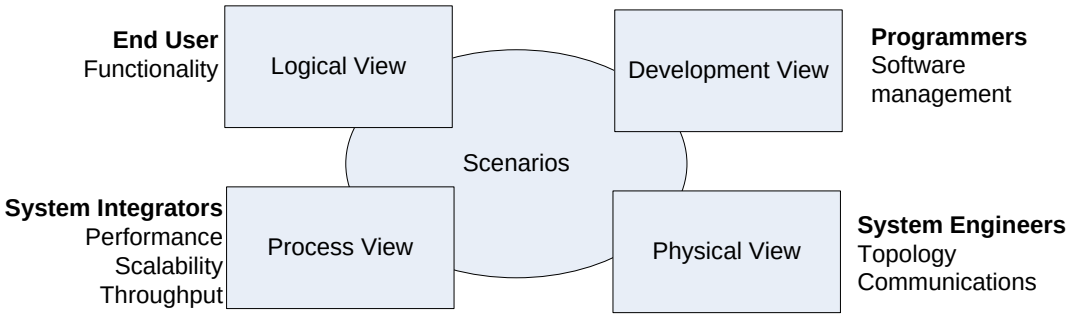


Figure 38: 4+1 Architecture Views

- **Logical View** primarily supports the functional requirements – the services the system should provide to its end users. Designers decompose the system into a set of key abstractions, taken mainly from the problem domain. In addition to aiding functional analysis, decomposition identifies mechanisms and design elements that are common across the system.
- **Development View** focuses on the organization of the actual software modules in the software-development environment. The software is packaged in small chunks – program libraries or subsystems – that can be developed by one or more developers. The subsystems are organized in a hierarchy of layers each of which provides a narrow and well defined interface to the layers above it. The development view takes into account internal requirements related to ease of development, software management, reuse or commonality, and constraints imposed by the toolset or the programming language. The development view is represented by module and subsystem diagrams that show the system's export and import relationships.
- **Process View** takes into account some nonfunctional requirements, such as performance and system availability. It addresses concurrency and distribution, system integrity, and fault-tolerance. The process view also specifies which thread of control executes each operation of each class identified in the logical view. Designers describe the process view at several levels of abstraction, each one addressing a

different concern. A process is a group of tasks that form an executable unit. Processes represent the level at which the process view can be tactically controlled (started, recovered, reconfigured, shut down, and so on). In addition, processes can be replicated to distribute processing load or improve system availability.

- **Physical View** takes into account the system's non-functional requirements such as system availability, reliability (fault-tolerance), performance (throughput), and scalability. The software executes on a network of computers (the processing nodes). The various elements identified in the logical, process, and development views – networks, processes, tasks, and objects – must be mapped onto the various nodes. Several different physical configurations will be used – some for development and testing, others for system deployment at various sites or for different customers. The mapping of the software to the nodes must therefore be highly flexible and have a minimal impact on the source code itself.
- **Scenarios** are instances of more general use cases. The scenarios are in some sense an abstraction of the most important requirements. This view is redundant with the other ones (hence the "+1"), but it serves two main purposes: (1) as a driver to discover the architectural elements during the architectural design. (2) as a validation and illustration role after this architectural design is complete, both on paper and as the starting point for the tests of an architectural prototype.

**Conceptual, Module, Execution and Code Architectures**

*Software Architecture in Industrial Applications* [12] by D. Soni, R. L. Nord, and C. Hofmeister in 1995 came up with the similar idea. There are four architecture views as illustrated in Figure 39:

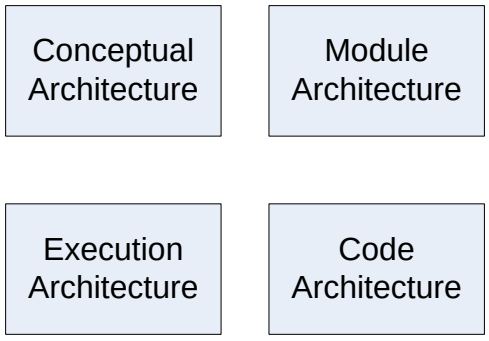


Figure 39: Conceptual, Module, Execution and Code Architectures

- **Conceptual Architecture** describes the system in terms of its major design elements and the relationships among them.
- **Module Architecture** encompasses two orthogonal structures: functional decomposition and layers.
- **Execution Architecture** describes the dynamic structure of a system.
- **Code Architecture** describes how the source code, binaries, and libraries are organized in the development environment.

**Module, Component-and-Connector and Allocation Structures**

In *Software Architecture in Practice* [10], L. Bass, P. Clements, and R. Kazman also defined three categories of structures or views as illustrated in Figure 40:

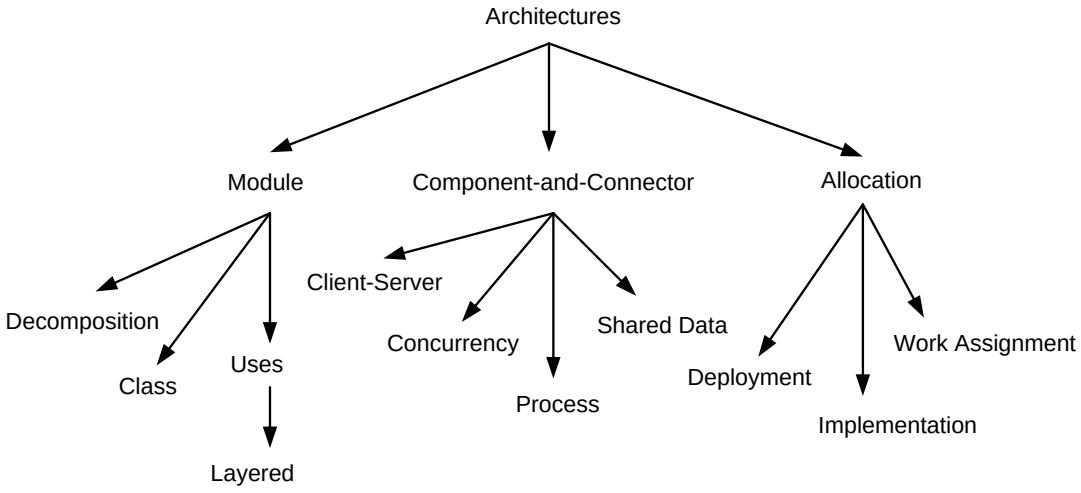


Figure 40: Module, Component and Connector, Allocation Structures

- **Module structures**. Here the elements are modules, which are units of implementation. Modules represent a code-based way of considering the system. They are assigned areas of functional responsibility. There is less emphasis on how the resulting software manifests itself at runtime. Module structures include Decomposition, Class or generalization and Uses structures. Layered structure is a special case of Uses structure.
- **Component-and-Connector structures**. Here the elements are runtime components (which are the principal units of computation) and connectors (which are the communication vehicles among components). Examples of this structure include Client-Server, Concurrency, Process and Shared Data.
- **Allocation structures**. Allocation structures show the relationship between the software elements and the elements in one or more external environments in which the software is created and executed. Work Assignment, Deployment and Implementation structures are in this category.

**Business, Solution and Deployment Blueprints**

*Enabling Iterative Software Architecture Derivation Using Early Non-Functional Property Evaluation* [4] by K. Suzanne Barber, Thomas Graser, et al introduced three architectures later called blueprints as illustrated in Figure 41:

- **Business Blueprint** defines the requirements in problem domain with minimal technology involvement.
- **Solution Blueprint** defines the solution of the problem. It attacks the problem with selected technology while satisfying non-functional requirements of the system.
- **Deployment Blueprint** defines the physical layout of the components from solution blueprint to suit certain non-functional requirements.

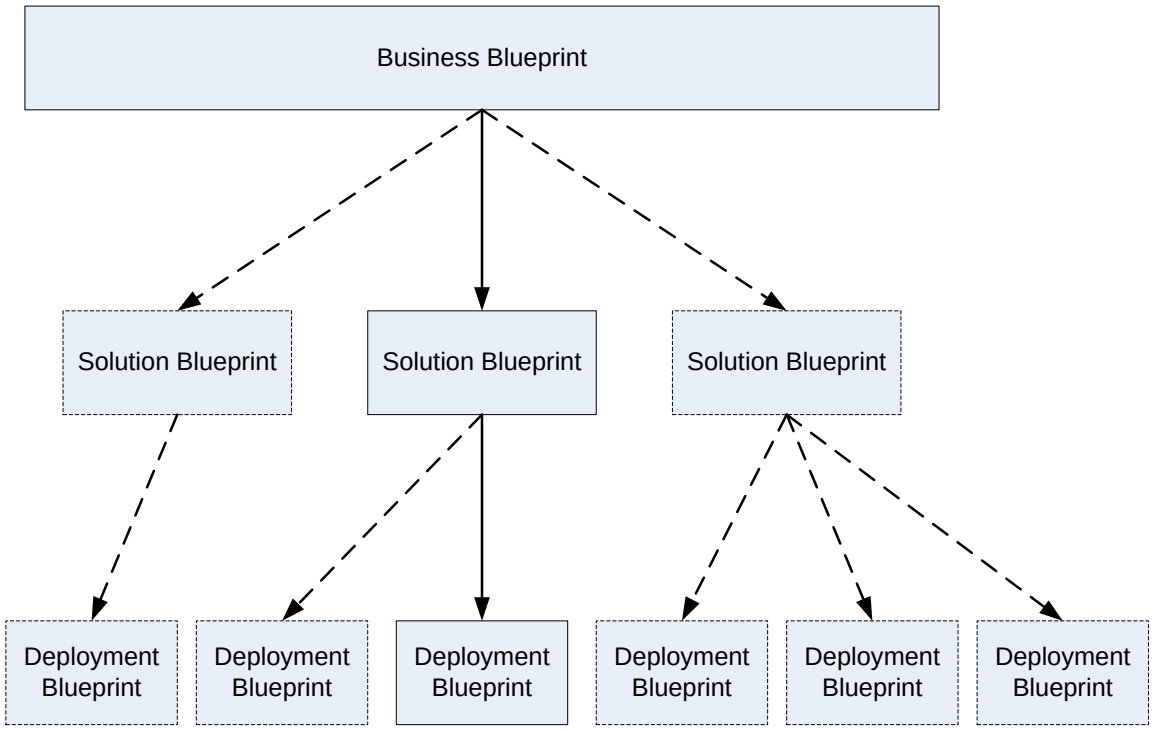


Figure 41: Business, Solution and Deployment Blueprints

As one problem may have multiple solutions each of which can be deployed in different ways, there are one-to-many relationship from Business Blueprint to Solution Blueprint and from Solution Blueprint to Deployment Blueprint. Figure 41 illustrates this relationship.

### DERIVATION

Four methodologies in representation section all try to categorize different views software architecture may have. There are many similarities among them. Logical View, Conceptual Architecture, Business Blueprint model the problem domain, or functional requirements of system stakeholders, or end users. Development View and Module Architecture capture static structures of the solution for the system while Process View, Execution Architecture and Component-and-Connector structure capture the dynamic aspect of the solution for the system. Both static and dynamic views are part of Solution Blueprint. Physical View and Allocation structures are deployments of the solution, or Deployment View.

Based on Business, Solution and Deployment Blueprint methodology and combining other three methods, we have a clear software architecture derivation workflow as shown in Figure 42. The process starts from identifying all stakeholders and collecting their functional as well as non-functional requirements and drawing use scenarios to clarify them. With architectural consideration designer derives Business Blueprint by presenting the functional requirements with logical views iteratively with stakeholders. After the problem is clearly defined a Solution Blueprint is derived from Business Blueprint with more considerations such as technology, development environment and libraries, team assignment as well as non-functional requirements. Solution Blueprint is represented by both static and dynamic views of the system as mentioned in the previous section. And finally architect presents Deployment Blueprint with site consideration and non-functional requirements in mind.

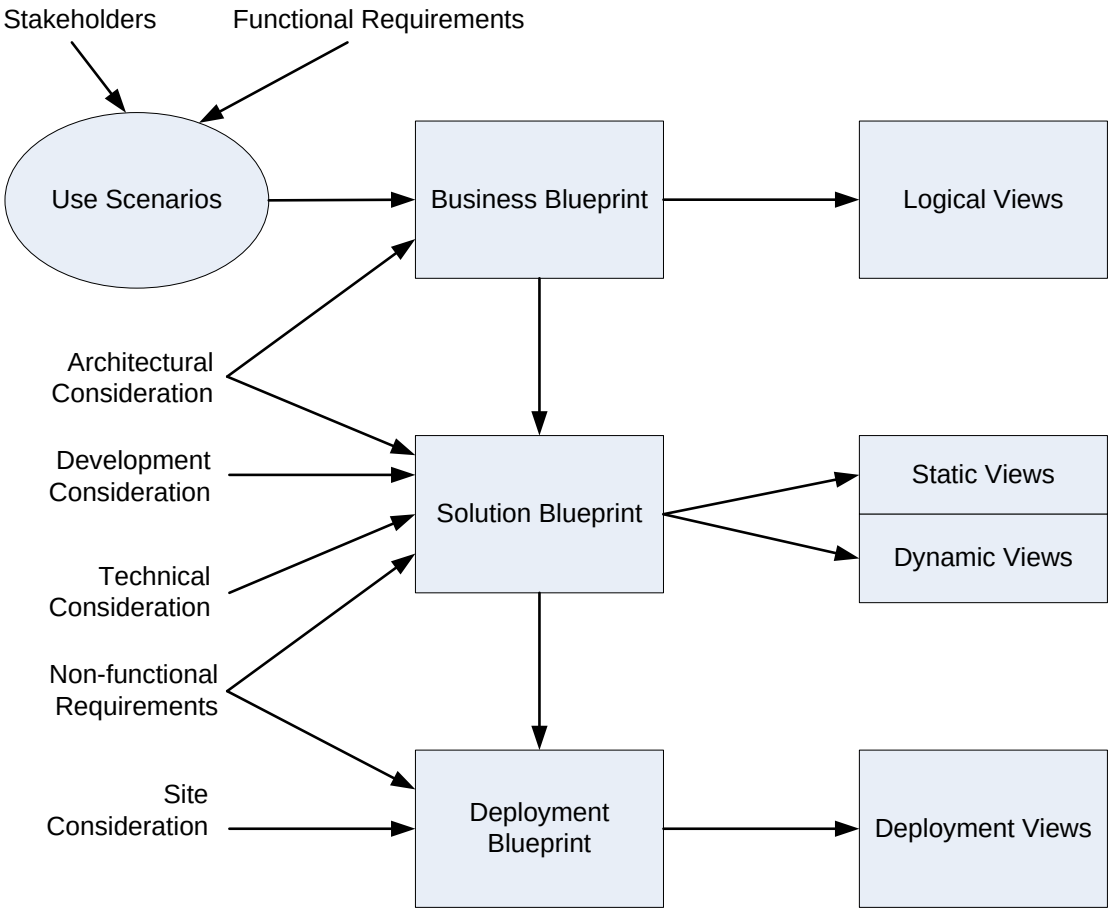


Figure 42: Software Architecture Derivation Workflow

# Bibliography


[1] Aldridge, Irene, 2009, *High-frequency trading: a practical guide to algorithmic strategies and trading system*. ISBN 978-0-470-56376-2

[2] Tharp, Van K., 2006, *Trade your way to financial freedom*. ISBN 0-07-147871-X.

[3] Smitten, Richard, 2001, *Jesse Livermore: world's greatest stock trader.* ISBN 0-471-02326-4.

[4] Barber, K.S., Graser, T., and Holt, J. 2002, *Enabling Iterative Software Architecture Derivation Using Early Non-Functional Property Evaluation*, Automated Software Eng. Proc. ASE 2002. 17th IEEE Int'l Conf. 2002.

[5] Malkiel, B.G, 2002, *A random walk down Wall Street*. ISBN 0-393-32535-0

[6] Perry, D. E. and Wolf, A. L. 1992, *Foundations for the Study of Software Architecture.* ACM Software Eng. Notes, vol. 17, no. 4 1992, pp. 40-51.

[7] Garlan, D. and Shaw, M. 1993, *An Introduction to Software Architecture*, Advances in Software Engineering and Knowledge Engineering, Vol. I.

[8] Hayes-Roth, F. 1994, *Architecture-Based Acquisition and Development of Software: Guidelines and Recommendations from the ARPA Domain-Specific Software Architecture (DSSA) Program.* Technical Report. Teknowledge Federal Systems, Palo Alto, CA.

[9] Gracek, C., Abd-Allah, A. Clark, B. Boehm, B, 1995, *On the Definition of Software System Architecture*. Technical Report, USC/CSE-95-TR-500

[10] Bass, L., Clements, P., and Kazman, R. 2003, *Software Architecture in Practice*, Addison-Wesley.

[11] Kruchten, P. 1995, *Architectural Blueprints - The 4+1 View Model of Architecture*, IEEE Software, vol. 12, no. 6, 1995, pp.45-50.

[12] Soni, D., Nord, R., and Hofmeister, 1995, C. *Software Architecture in Industrial Applications*, Proc. 17th Int'l Conf. Software Eng. (ICSE-17), ACM Press, 1995, pp. 196-207.

[13] Ye, Gewei, 2010, *High-frequency trading models.* ISBN 978-0-470-63373-1

[14] Narang, Rishi K, 2009, *Inside the black box: the simple truth about quantitative trading*. ISBN 978-0-470-43206-8

[15] Miles, R. and Hamilton, K., 2006, *Learning UML 2.0*, ISBN 0-596-00982-8

[16] C++ vs. Python vs. Perl vs PHP performance benchmark, http://blog.famzah.net/2010/07/01/cpp-vs-python-vs-perl-vs-php-performance-benchmark/

[17] TA-Lib: Technical Analysis Library, http://ta-lib.org

# Vita

Jinxiang Lu currently works with Kershner Trading Group to build proprietary trading platforms. Before that he worked for Questrade, a retail brokerage in Toronto, Canada. He worked on network applications in both desktop and embedded systems with SiliconOptix, a video chip startup and Microsoft. He enrolled in Software Engineering graduate program at the University of Texas at Austin in 2010 with a focus on distributed systems.

Email Address: jasonlue@hotmail.com

This report was typed by Jinxiang Lu.